TAPIDO: Trust and Authorization via Provenance and Integrity in Distributed Objects[⋆]

Andrew Cirillo, Radha Jagadeesan, Corin Pitcher, and James Riely

School of CTI, DePaul University

Abstract. Existing web services and mashups exemplify the need for flexible construction of distributed applications. How to do so securely remains a topic of current research. We present TAPIDO, a programming model to address Trust and Authorization concerns via Provenance and Integrity in systems of Distributed Objects. Creation of TAPIDO objects requires (static) authorization checks and their communication provides fine-grain control of their embedded authorization effects. TAPIDO programs constrain such delegation of rights by using provenance information. A type-and-effect system with effect polymorphism provides static support for the programmer to reason about security policies. We illustrate the programming model and static analysis with example programs and policies.

1 Introduction

Web services, portlets, and mashups are collaborative distributed systems built by assembling components from multiple independent web applications. Building such systems requires programming abstractions that directly address service composition and content aggregation. From a security standpoint, such composition and aggregation involves subtle combinations of authentication, authorization, delegation, and trust.

The issues are illustrated by account aggregation services that provide centralized control of an individual's accounts held with one or more institutions. An individual first grants permission for an aggregator to access owned accounts located at various institutions. In a typical use case, the aggregator is asked to provide a summary balance of all registered accounts: the aggregator asks each institution for the relevant account balance; the institution then determines whether or not to grant access; with the accumulated balances, the aggregator returns a summary of registered accounts to the individual. This simple service already raises several security and privacy issues related to trust and authorization. To name just two:

– The account owner's intent to access their account should be established by the institution. Message integrity is required to verify such intent.
– Principals should establish that the flow of messages through the system complies with authorization, audit, and privacy policies for account access. Message provenance is required to verify that the message history does comply with such policies.

⋆ Andrew Cirillo and James Riely were supported by NSF Career 0347542. Radha Jagadeesan and Corin Pitcher were supported by NSF Cybertrust 0430175.

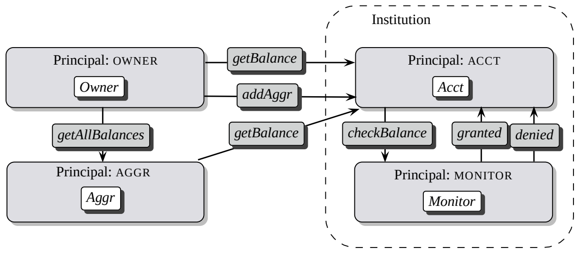

Fig. 1. Principals Involved in Account Aggregation

It has been said that "An application can be mashup-friendly or it can be secure, but it cannot be both." [1]. We disagree. In this paper, we describe the use of message provenance and integrity to achieve both security and flexibility aims in this general programming context.

In the remainder of this section, we present an informal overview of our approach using the account aggregation example. The principals involved are the account owner, the aggregation service, and two principals for the institution holding the account. The institution uses two principals to distinguish privileged monitor code from public-facing, unprivileged code. The owner requests the balance from the public-facing account object, which in turn contacts a trusted monitor to determine whether access should be granted or denied. The flow of messages is summarized in Figure 1.

Object model. TAPIDO's object model is based upon Java's notion of remote objects. We locate objects at atomic principals. Examples of atomic principals are nodes on a distributed system, a user or a process. For an object p, the location is available to the programmer via $p.\mathsf{loc}$. As with Java's remote objects, objects are immobile and rooted at the location where they are created. 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. References to objects are mobile — they can be freely copied and they move around through the system as arguments to methods or return values. We do not address mobility of objects themselves; thus, we do not discuss serialization and code mobility.

TAPIDO assumes a communication model that guarantees the provenance and integrity of messages. Thus, TAPIDO focuses on semantic attacks on trust and authorization, rather than on attacks against the cryptographic techniques required to achieve this communication model. Thus, our approach assume an underlying network model in which the sender of the message can be reliably determined; this model is well-studied [2,3,4,5] and realizable [6,7,8]. Using a relatively high-level model permits us to concentrate on attacks that seek unauthorized access, rather than studying the underlying cryptographic protocols that facilitate the integrity assumption.

Statics. Effects are communicated through object references. The language of effects is a decidable monotonic fragment of first-order logic (e.g., Datalog) extended to work over authorization logics. The modalities of authorization logics [9,10,11,12] permit different participants of a distributed system to maintain potentially inconsistent world-views, e.g. if *b* receives an object with effect ϕ created by *a*, it receives the effect *a says* ϕ, rather than the more absolute truth ϕ. Our language of effects also includes logic variables to achieve ML-style polymorphism with respect to effects.

Our "object-centric" notion of effects differs from the more usual "method-centric" notions explored in the literature on effects in Object-Oriented (OO) languages. The effects on objects can only refer to the immutable data of the object — if the object is an authorization token, this effect can record the rights associated with these object. For honest agents, object effects are validated at the point of creation, effectively ensuring that the global policy permits the creation of the object. When such an object is received — e.g., as an argument to a method call — the effects are transferred as a benefit to the recipient. In any execution of a well-typed program, there is a *corresponding* [13] object creation validating such accrual of rights.

The attackers that we consider are untrustworthy atomic principals running *any* well-typed Java program. Following [14] and our own earlier work [15], they may "utter" anything whatsoever in terms of effects. For example, opponents may create authorization objects without actually having the rights to create them, aiming to subvert the global authorization policy. A program is *safe* [16] if every object creation at runtime is justified by the accumulated effects. Our type system ensures that well-typed programs remain safe under evaluation in the face of arbitrary opponent processes.

In the account aggregation example, consider when an individual requests their balance from the institution holding their account through the aggregator. The guarantee sought is that the institution may only respond with the account balance when the request is approved by the account owner. With a pre-arranged protocol, approval can be conveyed by a message passed from the account owner to the institution via the aggregator. The institution's code must be able to verify that it originates with the owner and not been modified en route. The code must also ensure that the integrity-verified message and the pre-arranged protocol entail the owner's approval in the past; even in the presence of attackers who (perhaps falsely) claim possession of rights.

We describe a program incorporating such a design in our model, and verify the required properties with our static analysis.

Programming Provenance. Provenance — the history of ownership of an object — has received much interest in databases, e.g., see [17] for a survey. Security-passing style implementations [18] of stack inspection are already reminiscent of such ideas in a security context, since the provenance of the extra security-token parameter can be viewed as encoding the current relevant security context.

Provenance plays a crucial role in both the privacy architecture and the security (access control and accountability) of the account aggregation example. Consider the request from the account owner to the institution via the aggregator. The institution may impose an access control policy on the provenance of the request, e.g., to restrict the aggregators that can be used with the institution's services. Such a policy is distinct from, but can be used in conjunction with, an access control policy based upon the originator of the request. Similarly, the institution's audit policy may require a record of the provenance of requests (including the identities of the owner and the aggregator) to support an accountability obligation, e.g., to explain why and to whom account information was provided should the institution be accused of dishonest behavior.

Finally, the account owner can demand security of the path traversed by the result of such a request to ensure data privacy. This is demonstrated to the account owner by returning the relevant snapshot of the history of their data along with their data.

In contrast to stack inspection and history-based access control (e.g., see [19]) that mandate the flow of the security token, and record in it the *full* history of information used to make a judgement, our "user-defined" approach relies on trust relationships between the principals that are recorded as part of the history to make judgements.

In the account aggregation example, the response from the institution to the account owner has full history that can be described with the regular expression $\text{ACCT} \cdot \mathit{trusted}^* \cdot \text{AGGR} \cdot \mathit{trusted}^* \cdot \text{OWNER}$, where *trusted* represents a collection of trusted principals. Our explicit programming of this path in the sequel maintains only a subsequence of the history that matches $\text{ACCT} \cdot \mathit{trusted}^* \cdot \text{AGGR} \cdot \text{OWNER}$. Such abbreviations of the full history are codified in the security policy by assumptions on these principals — e.g., that the aggregator received the result from a trustworthy principal that can be relied upon to enforce the policy, *and* that the aggregator can be relied upon to report this information accurately.

We describe a program incorporating such a design in our model, and verify the required properties with our static analysis.

Related work. The study of effect systems was initiated in the context of functional languages (e.g., see Gifford and Lucassen [20,21], and Talpin and Jouvelot [22,23] amongst others). The ideas have since been applied broadly to OO languages; to name but a few, specifying the read/write behavior of methods [24,25], confinement [26,27], type reclassification [28], object protocols [29] and session types [30].

The most closely related papers are types for authorization, by Fournet, Gordon and Maffeis [31], a successor paper by the same authors [14] and our own earlier paper [15]. All of these papers (including this one) focus on authorization issues and so the work on information flow, e.g., see [32] for a survey, is not directly relevant. However, as in information flow based methods, TAPIDO global policy drives program design.

Fournet, Gordon and Maffeis [31] introduce an assume-guarantee reasoning framework with Datalog assertions for dealing with types for authorization. Both papers [31,14] are based in a pi-calculus formalism and view authorization as "a complex cryptographic protocol" [31] in the context of the traditional "network is the opponent" model. The successor paper uses dependency analysis on authorization logic to formalize a subtle notion of security despite compromise. Our object-centric effects adapt their static annotations to an OO setting. Our requirements on object creation (resp. transfer of effects to the callee) are analogous to their *expectation* (resp. *statement*) annotations.

Our prior paper [15] was inspired by [31]. It was also placed in a mobile process calculus, but diverged from [31,14] in assuming a model with explicit identities and a network that guaranteed integrity.

In this paper, we study imperative distributed objects by building on these intuitions. Our primary aim in this paper is to provide foundations of a programming methodology to ensure that distributed systems validate authorization and security policies; e.g., one of the aims of our examples is to illustrate the use of standard OO mechanisms to incrementally construct security guarantees. While the pi-calculus (with notions of keys) is expressive enough to code distributed objects (with explicit identities), such a translation is arguably inconsistent with our overall aims — just consider the complex encoding of state in the control of a pi-program. Such a translation based semantics approach obfuscates the simple (from an object standpoint) invariants that underlie our analysis. At any rate, the type systems in these three papers do not include the invariants of processes required to capture the type annotations of TAPIDO.

2 Language

We present the evaluation semantics for TAPIDO, a distributed class-based language with mutable objects. Our treatment of classes follows earlier direct semantics for class-based languages [33,34,24,35]. We do not address issues of genericity [36,34] or inner classes [37]. Our treatment of concurrency follows Gordon and Hankin's concurrent object calculus [38]. As in Cardelli's Obliq [39], our object references have distributed scope, rather than local scope [40]. Our treatment of locations borrows heavily from process algebras with localities (see [41] for a survey).

We first describe our naming conventions. 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 names "this" and "caller"; the binary predicate constructors "$\wedge$", representing conjunction, and "says", representing quoting; the ternary predicate constructor *Prov* is used to indicate that the first argument (an object) was received from the second argument (source principal) by the third argument (target principal). We write the 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.

$C,D ::= c\texttt{<}\vec{\phi}\texttt{>}$	Object Types
$T,S ::= C \mid \mathsf{Prin} \mid \mathsf{Unit}$	Value Types
$P,Q ::= \mathsf{Pred}(\vec{\mathscr{T}})$	Predicate Types
$\mathscr{T},\mathscr{S} ::= T \mid P \mid \mathsf{Proc}$	Types
$\mu ::= \mathsf{final} \mid \mathsf{mutable}$	Mutability Annotations
$\mathscr{D} ::= \mathsf{class}\ c\texttt{<}\vec{\alpha}:\vec{P}\texttt{>}\triangleleft D\{\vec{\mu}\ \vec{T}\ \vec{f};\ \vec{\mathscr{M}}\}\,[\theta]$	Classes ($\vec{\alpha}$ bound in D, θ, $\vec{T}$, $\vec{\mathscr{M}}$)
$\mathscr{M} ::= \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\ \ell(\vec{T}\ \vec{x})\{M\}$	Methods ($\vec{\beta}$ bound in S, $\vec{T}$, M; $\vec{x}$ in M)

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 θ (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, which are used in the statics. The syntax of values and terms is as follows[1].

V,W,U,A,B,ϕ,ψ ::=	Open Extended Values
$x \mid p \mid a \mid \mathsf{unit}$	Variable, Runtime Value
$\alpha \mid \gamma \mid \phi(\vec{V}) \mid \cdots$	Predicates
M,N,L,θ ::=	Terms
$V \mid \mathsf{new}\ c\texttt{<}\vec{\phi}\texttt{>}(\vec{V})$	Value, Object Creation
$\mathsf{let}\ x = V.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W})\,;M \mid V.f \mid V.\mathsf{loc} \mid V.f := W$	Object Operations
$\mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{let}\ x = N;\ M \mid N \parallel M$	Control Flow
$p : c\{\vec{f} = \vec{V}\} \mid (\nu p : C)\, M \mid a[M]$	Runtime Terms

We use the metavariables ϕ, ψ and θ 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, which do not play any role in the dynamics.

An *expectation* "expect θ" may be written as "new Proof<θ>()", where class Proof is defined "class Proof<α : Pred>{}[α]".

The syntax of terms includes standard OO primitives for object creation, method call, and field get/set. The let binder in method calls is necessary to describe the provenance of return values. Constructors and methods take predicate parameters that are used statically. The special "field" loc returns the location of an object. The conditional allows equality testing of values.

Concurrent composition ($\parallel$) is asymmetric. In $N \parallel M$, the returned value comes from M; the term N is available only for side effects. In the sequential composition "let $x = N;\ M$", x is bound with scope M. We elide the let, writing simply "$N;\ M$" when x does not occur in M. We also use standard syntactic sugar in place of explicit sequencing. For example, we may write "$y.f.g$" to abbreviate "let $x = y.f;\ x.g$".

Heap elements ($p : c\{\cdots\}$), name restriction ((νp)) and frames ($a[M]$) are meant only to occur at runtime. The first two of these model the heap, whereas the last models the (potentially distributed) "call stack". We expect that these constructs do not occur in user code. An object name binder (ν) is separate from the associated denotation ($p : c\{\vec{f} = \vec{V}\}$), allowing arbitrary graphs of heap objects. (The preceding example indicates that p is located at a, with actual class c and fields $\vec{f} = \vec{V}$.) The frame $a[M]$ indicates that M is running under the authority of a.

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

[1] When writing definitions using classes and methods, we often elide irrelevant bits of syntax, e.g., we leave out the parameters to classes when empty, such as writing Object rather than Object<·>. We identify syntax up to renaming of bound names, and write $M[x := V]$ for substitution of V for x in M (and similarly for other categories). We sometimes write extends for $\lhd$ for clarity. We often elide type information. We write "$S\ \ell\ (\vec{T}\ \vec{x});$" as shorthand for "$S\ \ell\ (\vec{T}\ \vec{x})\ \{\}$".

in the left column are from [38]. They capture properties of concurrent composition, including semi-associativity and the interaction with let. The rules in the right column, inspired by [41], capture properties of distribution. The first of these states that the interpretation of a value is independent of the location at which it occurs. The second states that computation of a frame does not depend upon the location from which the frame was invoked.

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

$(M \Vdash N) \Vdash L \equiv M \Vdash (N \Vdash L)$	$a[V] \equiv V$
$(M \Vdash N) \Vdash L \equiv (N \Vdash M) \Vdash L$	$a[b[M]] \equiv b[M]$
$((\nu p)\, N) \Vdash M \equiv (\nu p)(N \Vdash M)$	$a[N \Vdash M] \equiv a[N] \Vdash a[M]$
$M \Vdash ((\nu p)\, N) \equiv (\nu p)(M \Vdash N)$	$a[(\nu p)\, N] \equiv (\nu p)\, a[N]$
$\mathsf{let}\ x = (L \Vdash N);\, M \equiv L \Vdash (\mathsf{let}\ x = N;\, M)$	$a[\mathsf{let}\ x = N;\, M] \equiv \mathsf{let}\ x = a[N];\, a[M]$
$\mathsf{let}\ x = ((\nu p)\, N);\, M \equiv (\nu p)(\mathsf{let}\ x = N;\, M)$	

One may view interesting terms as *configurations*, which we now define. A *store* Σ is a collection of distributed heap terms, $b_1[p_1 : c_1\{\cdots\}] \Vdash \cdots \Vdash b_m[p_m : c_m\{\cdots\}]$, where each p_j is unique. A *thread* is either a value or a term $a[M]$ that does not contain occurrences of a name restriction or heap term. (A value represents a terminated thread.) An *initial* thread is a term $a[M]$ such that M additionally contains no blocks. A *configuration* is a term of the form $(\nu \vec{p})(\Sigma \Vdash M_1 \Vdash \cdots \Vdash M_n)$, where each M_i is a thread. A configuration is *initial* if each of its threads is initial. Evaluation preserves the shape of a configuration up to structural equivalence: If M is a configuration and $M \to M'$ then M' is structurally equivalent to a configuration.

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*; we elide the standard definitions. Evaluation is defined using the following axioms; we elide the standard inductive rules that lift structural equivalence to evaluation ($M \to M'$ if $M \equiv N \to N' \equiv M'$) and that describe computation in context (for example, $b[M] \to b[M']$ if $M \to M'$). We discuss the novelties below.

Term Evaluation $(M \to M')$

$\mathsf{new}\ c(\vec{V}) \to (\nu p)(p : c\{\vec{f} = \vec{V}\} \Vdash p)$
  if $\mathit{fields}(c) = \vec{f}$ and $|\vec{f}| = |\vec{V}|$

$b[p : c\{\cdots\}] \Vdash a[\mathsf{let}\ y = p.\ell(\vec{W});\, L] \to b[p : c\{\cdots\}] \Vdash a[\mathsf{let}\ y = b[M'];\, L']$
  if $\mathit{body}(c.\ell) = (\vec{x})\{M\}$ and $|\vec{x}| = |\vec{W}|$
  where $M' = \mathit{Prov}(\vec{W}, a, b) \Vdash M[\mathsf{caller} := a][\mathsf{this} := p][\vec{x} := \vec{W}]$
  and $L' = \mathit{Prov}(y, b, a) \Vdash L$

$b[p : c\{\cdots\}] \Vdash p.\mathsf{loc} \to b[p : c\{\cdots\}] \Vdash b$

$b[p : c\{f = V \cdots\}] \Vdash p.f := W \to b[p : c\{f = W \cdots\}] \Vdash \mathsf{unit}$

$b[p : c\{f = V \cdots\}] \Vdash p.f \to b[p : c\{f = V \cdots\}] \Vdash V$

$\mathsf{if}\ V = V\ \mathsf{then}\ M\ \mathsf{else}\ N \to M$

$\mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N \to N$ if $V \neq W$

$\mathsf{let}\ x = V;\, M \to M[x := V]$

The rule for new creates an object and returns a reference to it; in the Gordon/Hankin formalism, the heap stays on the left, whereas the return value goes on the right. $p.\mathsf{loc}$ returns the location of p.

Method invocation happens at the callee site, and thus a new frame is introduced in the consequent $b\,[M']$. The provenance of the actual parameters is recorded in $\mathit{Prov}(\vec{W}, a, b)$, which is shorthand for $\mathit{Prov}(W_1, a, b), \ldots, \mathit{Prov}(W_n, a, b)$. In M', the special variable caller is bound to calling principal; there are also standard substitutions for this and the formal parameters. In L', the provenance of the return value is recorded in $\mathit{Prov}(y, b, a)$.

Effects. Effects play a crucial role in the statics, but are ignored by evaluation. In summary, trustworthy processes are required to justify object creation by validating the expectations associated with classes in terms of accumulated effects. Opponent processes, on the other hand, may ignore expectations but are otherwise well typed. We say that a term is *safe* if the expectations associated with object creation by trusted principals during evaluation are always justified by the accumulated effects. We establish the standard properties of Preservation and Progress. As a corollary, we deduce that well-typed trustworthy processes remain safe when composed with *arbitrary* opponents.

Our proof of type-safety identifies the key properties required of the logic of effects. Thus, the logic of effects has to support structural rules on the left, support transitivity via cut, and ensure closure of the equality predicate under substitution and reduction. In addition, typechecking of examples (such as the ones that follow) also requires closure of inference under the inference rules of affirmation in the authorization logic of [10], e.g., functoriality of *says*, distribution of *says* over conjunction, and $(\alpha \Rightarrow A\ \mathit{says}\ \beta) \Rightarrow (A\ \mathit{says}\ \alpha \Rightarrow A\ \mathit{says}\ \beta)$. The full type and effect system and results with proofs can be found in the appendix.

3 Examples

In these examples, effects are described in a variant of Datalog extended to work over authorization logic. As with regular Datalog, a program is built from a set of Horn clauses without function symbols. In contrast to regular Datalog, the literals can also be in the form of quotes of principals. The well-formed user predicates are typed, with fixed arity. They are always instantiated with pure terms in a type-respecting fashion; pure terms are guaranteed to converge to a value without mutating the heap.

3.1 Workflow.

In this stateful workflow pattern, a user submits data of type T by creating an object of class SubmittedCell. (For simplicity, we do not address generic types here.) The manager must subsequently approve the data by creating an object of class ApprovedCell.

```
class Celll<α,β:Pred(T)> { }
class SubmittedCell<α,β:Pred(T)> extends Celll<α,β> {
  final T data; final Prin user; final Prin manager;
} [this.user says α(this.data)]
```

```
class ApprovedCell<α,β:Pred(T)> extends CellI<α,β> {
  final T data; final Prin user; final Prin manager;
} [this.user says α(this.data) ∧ this.manager says β(this.data)]
class FailedCell<α,β:Pred(T)> extends CellI<α,β> { }
```

In CellI<α,β>, α is the predicate that the user establishes on the data in the submission. β is the predicate that the manager establishes on the data. The final effect on approved cells represents both approvals in the static types.

The submission and approval objects are generated by a CellFactory in response to receipt of a request object (of class CellReq<γ>). The submit method of CellFactory<α,β> receives the effect req.loc *says* α(req.data) on its req parameter. The resulting instance of SubmittedCell<α,β> carries this assumption, along with the name of a manager that must approve the request.

```
class CellReq<γ:Pred(T)> { final T data; } [γ(this.data)]
class CellFactory<α,β:Pred(T)> {
  SubmittedCell<α,β> submit(CellReq<α> req, Prin manager) {
    new SubmittedCell<α,β>(req.data, req.loc, manager)
  }
  CellI<α,β> approve(CellReq<β> req, SubmittedCell<α,β> cell) {
    if ((req.loc=cell.manager) && (req.data=cell.data) && (this.loc=cell.loc))
    then new ApprovedCell<α,β>(cell.data, cell.user, cell.manager)
    else  new FailedCell<α,β>()
} }
```

The approve method receives the effect req.loc *says* β(req.data). After checking that req.loc is the same as cell.manager, it may conclude that cell.manager *says* β(req.data). To establish the final effect on the ApprovedCell, the factory must establish that the data in the approval request is the same as the data in the initial request. Further, it must be the case that submit and approve are called upon factories located at the same principal, since the ApprovedCell vouches for both α and β, although these are validated at different times. If any of the equality tests are missing, the code fails to typecheck.

Visitors for typecases. The class CellI is an interface for cells. The visitor design pattern [42] provides a type-safe way to write code that is dependent on the actual dynamic type/subclass. Thus, we add methods such as visitApprovedCell to class CellV<α,β> (in general, one such visit method for each subclass). To dispatch to the visitor, the CellI interface is augmented with an accept method, implemented in each subclass; e.g., if S is the return type of the visitor, the implementation of ApprovedCell<α,β>.accept is:

```
S accept(CellV<α,β> v) { v.visitApprovedCell(this) }
```

Encoding Provenance. The submission and approval requests described above for the workflow cell do not track provenance. To accommodate provenance tracking, e.g., for the account balance requests discussed in Section 1, we develop an idiom for decorating such requests as they are passed from principal to principal. The decorations indicate the provenance of the transmitted data. As usual with a decorator design pattern [42], the Req<α> class is split into three classes: the interface ReqI<α>, the concrete class

ReqC<α> (which corresponds to the original Req<α>), and the decorator ReqD<α>. We use a visitor to inspect the resulting object. Again, let T be the type of the request data and S be the arbitrary return type of the visitor.

```
class ReqV<α> { S visitReqC(ReqC<α> x); S visitReqD(ReqD<α> x); }
class ReqI<α> { S accept(ReqV<α> v); }
class ReqC<α> extends ReqI<α> { final T data;
  S accept(ReqV<α> v) { v.visitReqC(this) }
} [α(this)]
class ReqD<α> extends ReqI<α> { final ReqI<α> payload; final Prin src; final Prin tgt;
  S accept(ReqV<α> v) { v.visitReqD(this); }
} [Prov(this.payload, this.src, this.tgt)]
```

Significantly, it is the concrete class ReqC<α> that retains the original effect α(this). The decorator, instead, carries an effect concerning the provenance of the decorated data. The effect *Prov*, used here at type Pred(ReqI<α>, Prin, Prin), is a claim about the provenance of one hop of a request. It indicates that this.payload was received from this.src by this.tgt. Thus, the object creation **new** ReqD(p, A, B) typechecks only when the static semantics can deduce that p has been received by B from A.

To illustrate request decoration, consider the following trustworthy forwarder[2]:

```
class TrustworthyForwarder extends AggrI { mutable AggrI next;
  RespI getAllBalances(ReqI<SubmitBal> req) {
    let resp:RespI = next.approve(new ReqD<SubmitBal>(req, caller, this.loc));
    new RespD(resp, next.loc, this.loc); } }
```

The method body is typechecked in the context of the assertion *Prov*(req, caller, this.loc), thus permitting the construction of the ReqD object. Similarly, the *Prov*(resp, next.loc, this.loc) assertion established by the method invocation on next enables the typechecking of the construction of the new RespD object. In contrast, an untrustworthy forwarder might produce an inaccurate provenance decoration for the request, e.g., using **new**ReqD<*SubmitBal*>(req, FAKESRC, FAKETGT)). In the following account aggregation example, the principals trusted to provide accurate provenance decorations are specified via the θ_2 component of the global policy.

3.2 Account Aggregation.

Recall, from Figure 1, a rough outline of the protocol: (1) OWNER informs ACCT that AGGR may aggregate its balances (using Acct.addAggr); (2) OWNER requests a summary of its balances from AGGR (using Aggr.getAllBalances); (3) AGGR requests the balance from ACCT using Acct.getBalance. Steps (1) and (3) involve communication between the public-facing ACCT and the private MONITOR. In addition, let the principal FORWARDER be trusted to relay messages using the decorator previously discussed.

[2] For reasons of space we omit definition of AggrI, an interface class with a single getAllBalances method, and classes RespI, RespC, RespD for responses by analogy with non-generic versions of request classes ReqI, ReqC, ReqD.

For simplicity, we use a single forwarder and account as well as a single class to represent the code running at each principal. (We follow the convention that field owner references an instance of class Owner located at principal OWNER.) Due to space limitations, we elide the code implementing step (1) of the protocol. We recall that Step (2) of the protocol is initiated by the OWNER, with a call to Aggr.getAllBalances.

The global security policy. The global system policy has the form $[\text{OWNER } \textit{says } (\theta_0)] \wedge [\text{AGGR } \textit{says } (\theta_1 \wedge \theta_2 \wedge \theta_3)] \wedge [\text{MONITOR } \textit{says } (\theta_4 \wedge \theta_5)] \wedge [\text{ACCT } \textit{says } \theta_6]$. The predicates $\theta_0 \ldots \theta_6$ are formalized shortly. Informally, θ_0 will ensure that the OWNER is authorized to submit balance requests. θ_1 and θ_2 will characterize the paths that are considered secure. θ_3 will ensure that the aggregator only creates requests that arrive from owner on secure paths. θ_4 and θ_5 will ensure that the MONITOR only accepts requests from owner or from aggregators certified by the owner. θ_6 will ensure that the account delegates authorization decisions to the monitor.

The design of the entire program that follows is driven by this global policy, i.e., our code is set up to satisfy the expectations of each principal. Our presentation of the formal policies piecemeal along with the associated classes is only for concise exposition.

Notation. To encode the policy, we use several predicate constructors, which we write in italics. *SubmitAggr*, with type Pred(Prin), indicates that an aggregator has been submitted for approval. Likewise *ApproveAggr* indicates that the request was approved. *SubmitBal*, with type Pred(ReqC<*SubmitBal*>), is a claim that a balance request has been submitted. *ApproveBal*, with type Pred(ReqI<*SubmitBal*>), is a claim that a balance request (perhaps with decorators) has been approved. As described previously, *Prov*, used here at type Pred(ReqI<*SubmitBal*>, Prin, Prin), is a claim about the provenance of one hop of a request. *CheckedProv*, with type Pred(ReqI<*SubmitBal*>), indicates that the provenance of a request has been checked, and is specified using reachability via *Prov*, incorporating trust in principals that report about each hop.

We assume that the field Monitor.cell is set appropriately. For simplicity, we have hard-coded AGGR and other principals throughout the example code; one may instead use a final field to store principals of interest, deferring the choice to instantiation-time.

Owner. We use some abbreviations and elide the code to check the response received back from the aggregator, which is similar to the visitor used by the aggregator, shown later below. Acct.addAggr expects arguments of type CellReq<*SubmitAggr*>, and Aggr.getAllBalances expects arguments of type ReqI<*SubmitBal*>.

```
class Owner { mutable AcctI acct; mutable AggrI aggr;   /* could be forwarders */
 Unit main() {
    acct.addAggr(new CellReq<SubmitAggr>(AGGR));
    let response:RespI = aggr.getAllBalances(new ReqC<SubmitBal>(this.loc));
    . . . /* check response for compliance with privacy policy */ }
} [θ0]
```

where $\theta_0 = (\textit{SubmitAggr}(\text{AGGR})) \wedge (\textit{SubmitBal}(\mathbb{X})\,{:}\text{-}\,\mathbb{X}.\mathsf{data} = \mathbb{X}.\mathsf{loc} = \mathsf{this.loc})$. This effect indicates that the instantiator must be able to submit the aggregator request and that the instantiator must be able to submit any balance request that it creates, so long as the data field truthfully records its identity. The second requirement is expressed using a Datalog variable $\mathbb{X}$, ranging over values of type ReqC<*SubmitBal*>.

Aggregator. The code uses the following effects.

$\theta_1 = \mathit{CheckedProv}(\mathbb{X})$:- $\mathit{Prov}(\mathbb{X}, \mathbb{S}, \mathsf{this.loc})$, $\mathbb{S} = \textsc{owner}$ OR $\mathbb{S} = \textsc{forwarder}$
$\theta_2 = \mathit{CheckedProv}(\mathbb{X}.\mathsf{payload})$:- $\textsc{forwarder}$ *says* $\mathit{Prov}(\mathbb{X}.\mathsf{payload}, \mathbb{S}, \textsc{forwarder})$,
$\mathit{CheckedProv}(\mathbb{X})$
$\theta_3 = \mathit{SubmitBal}(\mathbb{X})$:- $\textsc{owner}$ *says* $\mathit{SubmitBal}(\mathbb{Y})$, $\mathbb{Y}.\mathsf{data}{=}\mathbb{X}.\mathsf{data}{=}\textsc{owner}$, $\mathit{CheckedProv}(\mathbb{Y})$

The first two of these deal with provenance. The base case θ_1 validates an object delivered to aggregator from forwarder or owner. θ_2 recurses down one level of the decorated object, making explicit the trust on trusted forwarders. Together θ_1 and θ_2 ensure that a request is deemed valid if it has passed through trusted intermediaries. θ_3 allows the aggregator to create new balance requests, if it has checked the provenance of the request: both the new request $\mathbb{X}$ and the old one $\mathbb{Y}$ must have the data field set to OWNER; further, the OWNER must avow that they created the old request.

```
class Aggr extends AggrI { final Acct acct;
  RespI getAllBalances(ReqI<SubmitBal> req) {
    if ((caller=FORWARDER) || (caller=OWNER)) then
      let req2:ReqI<SubmitBal> = req.accept(new AggrReqV(req));
      let resp:RespI = acct.getBalance(req2);
      new RespD(resp, acct.loc, this.loc) }
} [θ1 ∧ θ2 ∧ θ3]
```

The validation of the creation of req2 uses θ_1 to satisfy the effect of the the class AggrReqV. The auxiliary class AggrReqV is a visitor to typecase on the request being either a concrete request, or being a forwarded request.

```
class AggrReqV extends ReqV<SubmitBal> {
  final ReqI<SubmitBal> in;
  ReqI<SubmitBal> visitReqC(ReqC<SubmitBal> x) {
    if ((this.in=x) && (x.loc=x.data=OWNER)) then
      new ReqC<SubmitBal>(x.data)
    else ... /* error */ }
  ReqI<SubmitBal> visitReqD(ReqD<SubmitBal> x) {
    if ((this.in=x) && (x.loc=x.tgt=FORWARDER)) then
      x.payload.accept(new AggrReqV(x.payload))
    else ... /* error */ }
} [θ1 ∧ θ2 ∧ θ3 ∧ CheckedProv(this.in)]
```

As the visitor traverses the decorators, it maintains the invariant that *CheckedProv* is true of the object being visited. The visitor updates the effect each time it moves to a new element by creating (and using) a new visitor. On callback to visitReqC or visitReqD, the ReqI *should* be the same as the one with the effect; the type system ensures that this is explicitly checked. To type visitReqC requires θ_3, which allows us to create the new ReqC located at AGGR. To type visitReqD, we first deduce $\mathit{CheckedProv}(\mathsf{x})$ from $\mathsf{this.in} = \mathsf{x}$ and the class effect. Since x is a ReqD, we have x.loc *says Prov*(x.payload, x.src, x.tgt). Since x.loc = x.tgt = FORWARDER and *CheckedProv*(x), then θ_2 yields *CheckedProv*(x.payload), allowing creation of the new AggrReqV.

The enforcement of the privacy policy of the introduction by the OWNER can be achieved using similar techniques.

Account. Calls to Acct.getBalance are delegated to Monitor.checkBalance, which results in a call back to either Acct.granted or Acct.denied.

```
class Acct { mutable int Balance; mutable Monitor monitor; mutable RespI result;
  RespI getBalance(ReqI<SubmitBal> req) {
    monitor.checkBalance(req, this);
    this.result }
  Unit granted(ReqI<ApproveBal> req) {
    if (req.loc=MONITOR) then
      expect MONITOR says ApproveBal(req);
      this.result := new RespC(req)
    else . . . /* error */ }
  Unit denied() { . . . /* error */ } . . .
} [θ6]
```

Here $\theta_6 = \mathit{ApproveBal}(\mathbb{X})$:- MONITOR *says* $\mathit{ApproveBal}(\mathbb{X})$. Thus, if the granted method is called back, then it must be the case that the monitor approved the request.

Monitor. The effects of the monitor code are expressed using the following predicates.

$\theta_4 = \mathit{ApproveBal}(\mathbb{X})$:- OWNER *says* $\mathit{SubmitBal}(\mathbb{X})$, $\mathbb{X}$.data=OWNER
$\theta_5 = \mathit{ApproveBal}(\mathbb{X})$:- OWNER *says* $\mathit{SubmitAggr}(\mathbb{Y})$, this.loc *says* $\mathit{ApproveAggr}(\mathbb{Y})$,
$\mathbb{Y}$ *says* $\mathit{SubmitBal}(\mathbb{X})$, $\mathbb{X}$.data=OWNER

```
class Monitor { mutable CellI<SubmitAggr, ApproveAggr> cell;
  Unit checkBalance(ReqI<SubmitBal> req, Acct acct) {
    if (req.loc=req.data=OWNER)
    then /* audit the request */; acct.granted(new ReqC<ApproveBal>(req.data))
    else  this.cell.accept(new MonitorCellV(req, acct)) }
} [θ4 ∧ θ5]
class MonitorCellV extends CellV<SubmitAggr, ApproveAggr>  {
  final ReqI<SubmitBal> req; final Acct acct;
  Unit visitFailedCell(FailedCell<SubmitAggr, ApproveAggr> x) { this.acct.denied() }
  Unit visitSubmittedCell(SubmittedCell<SubmitAggr, ApproveAggr> x) { this.acct.denied() }
  Unit visitApprovedCell(ApprovedCell<SubmitAggr, ApproveAggr> x) {
    if ((x.loc=this.loc) && (OWNER=x.user) && (this.loc=x.manager)
        && (this.req.loc=x.data) && (this.req.data=OWNER))
    then /* audit the request */;  this.acct.granted(new ReqC<ApproveBal>(this.req.data))
    else  this.acct.denied() }
} [θ5]
```

In checkBalance, θ_4 establishes the safety of creating the ReqC, whereas θ_5 establishes the safety of creating the MonitorCellV.

4 Conclusion

TAPIDO is designed to counter the claim that "an application can be mashup-friendly or it can be secure, but it cannot be both." Our model of dynamics adds only two non-standard features, namely (a) the ability to detect the creator location, and (b) integrity of remote method invocation. We have shown that this suffices to code useful tracking of the provenance of an object reference. Our type system adds (polymorphic) object level effects to standard types. From a programming point of view, this style allows trust-based decisions that are validated by the policy context of the application.

References

1. Chess, B., O'Neil, Y.T., West, J.: Javascript hijacking. Technical report, Fortify Software (2007) http://www.fortifysoftware.com/news-events/releases/2007/2007-04-02.jsp.
2. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: theory and practice. ACM Trans. Comput. Syst. **10**(4) (1992) 265–310
3. Wobber, E., Abadi, M., Burrows, M., Lampson, B.: Authentication in the Taos operating system. ACM Trans. Comput. Syst. **12**(1) (1994) 3–32
4. Abadi, M., Fournet, C., Gonthier, G.: Authentication primitives and their compilation. In: POPL. (2000) 302–315
5. Landau, S.: Liberty ID-WSF security and privacy overview. http://www.projectliberty.org/ (2006)
6. Li, N., Mitchell, J.C., Tong, D.: Securing Java RMI-based distributed applications. In: ACSAC, IEEE Computer Society (2004) 262–271
7. Scheifler, B., Venners, B.: A conversation with Bob Scheifler, part I, by Bill Venners. http://www.artima.com/intv/jinisecu.html (July 2002)
8. Gordon, A.D., Pucella, R.: Validating a web service security abstraction by typing. Formal Asp. Comput. **17**(3) (2005) 277–318
9. Abadi, M., Burrows, M., Lampson, B.W., Plotkin, G.D.: A calculus for access control in distributed systems. ACM Trans. Program. Lang. Syst. **15**(4) (1993) 706–734
10. Abadi, M.: Access control in a core calculus of dependency. In: ICFP, ACM (2006) 263–273
11. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. CSFW **0** (2006) 283–296
12. Garg, D., Bauer, L., Bowers, K.D., Pfenning, F., Reiter, M.K.: A linear logic of authorization and knowledge. In: ESORICS. (2006) 297–312
13. Woo, T.Y., Lam, S.S.: A semantic model for authentication protocols. In: IEEE Symposium on Research in Security and Privacy. (1993)
14. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization in distributed systems. In: CSF, IEEE (2007)
15. Cirillo, A., Jagadeesan, R., Pitcher, C., Riely, J.: Do As I SaY! programmatic access control with explicit identities. In: CSF, IEEE (2007)
16. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security **11**(4) (2003) 451–520
17. Buneman, P., Tan, W.C.: Provenance in databases. In: SIGMOD Conference, ACM (2007) 1171–1173
18. Wallach, D.S., Appel, A.W., Felten, E.W.: SAFKASI: a security mechanism for language-based systems. ACM Trans. Softw. Eng. Methodol. **9**(4) (2000) 341–378
19. Abadi, M., Fournet, C.: Access control based on execution history. In: Proc. Network and Distributed System Security Symp. (2003)

20. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: LISP and Functional Programming. (1986) 28–38
21. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL. (1988) 47–57
22. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. J. Funct. Program. **2**(3) (1992) 245–271
23. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. **111**(2) (1994) 245–296
24. Bierman, G., Parkinson, M., Pitts, A.: MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory (April 2003)
25. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: ECOOP, London, UK (1999) 205–229
26. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. TOPLAS (2007) To appear.
27. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Featherweight generic confinement. J. Funct. Program. **16**(6) (2006) 793–811
28. Damiani, F., Drossopoulou, S., Giannini, P.: Refined effects for unanticipated object re-classification: Fickle$_3$. In: ICTCS. Volume 2841 of LNCS., Springer (2003) 97–110
29. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: PLDI. (2001) 59–69
30. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A distributed object-oriented language with session types. Volume 3705 of LNCS. (2005) 299–318
31. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. In: ESOP. Volume 3444 of LNCS., Springer (2005) 141–156
32. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications **21**(1) (January 2003) 5–19
33. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: POPL. (1998) 171–183
34. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA. (1999)
35. Drossopoulou, S., Eisenbach, S., Khurshid, S.: Is the Java type system sound? Theory and Practice of Object Systems **5**(11) (1999) 3–24
36. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: OOPSLA. (1998) 183–200
37. Igarashi, A., Pierce, B.C.: On inner classes. Information and Computation **177**(1) (2002) 56–89
38. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: Reduction and typing. In: Proceedings HLCL'98, ENTCS (1998)
39. Cardelli, L.: A language with distributed scope. In: POPL, ACM Press (1995) 286–297
40. Jeffrey, A.S.A.: A distributed object calculus. In: Proc. Foundations of Object Oriented Languages. (2000)
41. Castellani, I.: Process algebras with localities. In: Handbook of Process Algebra. North-Holland (2001) 945–1045
42. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
43. Fairtlough, M., Mendler, M.: Propositional lax logic. Inf. Comput. **137**(1) (1997) 1–33
44. Tse, S., Zdancewic, S.: Translating dependency into parametricity. In: ICFP, ACM (2004) 115–125
45. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**(3) (1997) 364–418
46. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. **17**(3) (1995) 507–535

A Background: Authorization Logics

We refer the reader to [11,10] for the intuitions underlying authorization logics. Our presentation satisfies more commutativity properties than [11] in the proof theory. In comparison to [10], we have no second-order quantifiers. This background section is drawn from our earlier paper [15].

The formulas are given by the following grammar: for expository purposes, we only consider conjunction & and implication $\rightarrow$.

$$\alpha, \beta \ ::= \ \mathsf{true} \ \mid \ \alpha \,\&\, \beta \ \mid \ \alpha \rightarrow \beta \ \mid \ A \textit{ says } \alpha$$

A *says* α connects the calculus of principals to the logic: this is the quoting combinator of the logic and is related to the quoting combinator of the lattice by defining $A|B$ *says* α to be A *says* B *says* α.

We describe Hilbert-style axioms, inspired by those for propositional lax logic [43], to describe the tautologies. We first define B-protected formulas [10,44]. Informally, if there is a proof of a B-protected formula, then there is one that does not require statements of principals that are more trustworthy than B.

Definition 1. *The class of B-protected formulas is defined inductively as follows: (a)* true *is B-protected. (b) A says* α *is B-protected if either* α *is B-protected. (c)* $\alpha \,\&\, \beta$ *(resp.* $\alpha \rightarrow \beta$*) is B-protected if* α *and* β *(resp.* β*) are B-protected.*

In concordance with the informal intuitions, the following axiom system satisfies the property that if a formula is B-protected and $A \Rightarrow B$, then the formula is also A-protected.

Definition 2. *The axioms of authorization logic (*$\vdash \alpha$*) are as follows.*

(a) Propositional validity: *If* α *is an instance of a intuitionist propositional tautology, then* $\vdash \alpha$.
(b) Modus Ponens: *If* $\vdash \alpha$ *and* $\vdash \alpha \rightarrow \beta$*, then* $\vdash \beta$.
(c) Modality-Unit: *If* $\vdash \alpha$*, then* $\vdash A$ *says* α
(d) Modality-Mult: *If* $\vdash \alpha \,\&\, \alpha' \rightarrow \beta$ *and* β *is B-protected, then* $\vdash (B \textit{ says } \alpha) \,\&\, \alpha' \rightarrow \beta$.

Following [10], examples of provable theorems are (a) Order Naturality: if $\vdash A$ *says* α and $A \Rightarrow B$, then B *says* α; (b) Reflexivity: A *says* A *says* $\alpha \leftrightarrow A$ *says* α; (c) Commutativity: A *says* B *says* $\alpha \leftrightarrow B$ *says* A *says* α; and (d) Extensivity: A *says* $\alpha \rightarrow B$ *says* A *says* α.

Remark 1. The primary use of principals in the logic is via the quoting formulas constructed with *says*. So, it is conceptually consistent to assume that properties (b)–(d) are reflected back into the security lattice, i.e., $|$ is reflexive, commutative, and extensive.

Extended Datalog. We describe a variant of Datalog extended to work over the authorization logic. In this discussion, for concreteness, we focus on extending positive Datalog — the same development works for more expressive fragments such as positive disjunctive Datalog [45].

As with regular Datalog, a program will be built from a set of Horn clauses without function symbols. In contrast to regular Datalog, the literals can also be in the form of quotes of principals.

The well-formed user predicates are typed and of fixed arity. They are always instantiated with pure terms in a type-respecting fashion. We will use binary predicates for quoting and equality, written respectively as A says ϕ and $V = W$. (We make liberal use of syntax sugar, more generally writing $M = N$.) The pieces of logic that occur in a program are extended Datalog programs that use such predicates.

We assume that the Datalog programs always contain the axioms required for = to be an equivalence, e.g. the clause for reflexivity is $x = x$:-; and congruence for each predicate in the program, e.g. for every 1-ary predicate γ, there is a clause $\gamma(x)$:- $\gamma(y), x = y$ as part of the Datalog program. We account for the logic variables by closing up the source program under all type-valid substitutions of predicates for logic variables.

Despite this extra generality, the extended formalism has decidable clause inference following [15] by a translation of extended Datalog into Datalog that is sound and complete for the inference of ground literals.

B Elided Definitions

We present several definitions elided from the main text.

Term Evaluation (Context Rules)

$$\frac{M \to M'}{b\,[M] \to b\,[M']} \qquad \frac{N \to N'}{\mathsf{let}\ x = N; M \to \mathsf{let}\ x = N'; M} \qquad \frac{M \equiv N \to N' \equiv M'}{M \to M'}$$

$$\frac{M \to M'}{M \mathbin{\|\!\!\ulcorner} N \to M' \mathbin{\|\!\!\ulcorner} N} \qquad \frac{N \to N'}{M \mathbin{\|\!\!\ulcorner} N \to M \mathbin{\|\!\!\ulcorner} N'} \qquad \frac{M \to M'}{(\nu p)\, M \to (\nu p)\, M'}$$

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

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

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

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

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

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

The typing system additionally uses a related function for predicate lookup, as well as a standard notion of well formed overriding. Recall that "$\theta_D \wedge \theta[\vec{\alpha} := \vec{\phi}]$" is sugar for "$\mathsf{let}\ x = \theta_D; \mathsf{let}\ y = \theta[\vec{\alpha} := \vec{\phi}]; x \wedge y$".

Predicate Lookup $(\mathit{effect}(C) = \theta)$

$$\frac{}{\mathit{effect}(\mathsf{Object}) = \mathsf{true}} \qquad \frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c{<}\vec{\alpha} : \vec{P}{>} \lhd D\{\cdots\}\,[\theta] \quad \mathit{effect}(D[\vec{\alpha} := \vec{\phi}]) = \theta_D}{\mathit{effect}(c{<}\vec{\phi}{>}) = \theta_D \wedge \theta[\vec{\alpha} := \vec{\phi}]}$$

Well Formed Overriding $(\vdash {<}\vec{\beta} : \vec{Q}{>}S(\vec{T})\ \mathit{can\ override}\ D.\ell)$

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

We now define a canonical form for terms up to structural equivalence. Let simple terms be defined as follows.

$$\begin{aligned}
\mathbb{L} \ ::=\ & \mathsf{new}\ c{<}\vec{\phi}{>}(\vec{V}) \mid V.\ell{<}\vec{\phi}{>}(\vec{W}) \mid V.f \mid V.\mathsf{loc} \mid V.f := W \\
& \mid \mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{let}\ x = a\,[\mathbb{L}]\,;M \mid \mathsf{let}\ x = V; M \\
\mathbb{N} \ ::=\ & \mathsf{new}\ c{<}\vec{\phi}{>}(\vec{V}) \mid V.\ell{<}\vec{\phi}{>}(\vec{W}) \mid V.f \mid V.\mathsf{loc} \mid V.f := W \\
& \mid \mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{let}\ x = \mathbb{N}; M \mid \mathsf{let}\ x = V; M \\
& \mid p : c\{\vec{f} = \vec{V}\}
\end{aligned}$$

Proposition 1. *For any term M there exists $M \equiv (\nu \vec{p} : \vec{C})(W_1 \Vdash \cdots \Vdash W_\ell \Vdash \mathbb{N}_1 \Vdash \cdots \Vdash \mathbb{N}_m \Vdash b_1\,[\mathbb{L}_1] \Vdash \cdots \Vdash b_n\,[\mathbb{L}_n] \Vdash M')$ such that M' has the form V or $\mathbb{L}$ or $a\,[\mathbb{N}]$; moreover, M' is unique.*

$$\mathit{right}(M) \triangleq \begin{cases} V' & \text{if } M \equiv (\nu \vec{p} : \vec{C})(\vec{W} \Vdash \vec{\mathbb{N}} \Vdash \vec{b}\,[\vec{\mathbb{L}}] \Vdash V') \\ \mathbb{N}' & \text{if } M \equiv (\nu \vec{p} : \vec{C})(\vec{W} \Vdash \vec{\mathbb{N}} \Vdash \vec{b}\,[\vec{\mathbb{L}}] \Vdash \mathbb{N}') \\ a\,[\mathbb{L}'] & \text{if } M \equiv (\nu \vec{p} : \vec{C})(\vec{W} \Vdash \vec{\mathbb{N}} \Vdash \vec{b}\,[\vec{\mathbb{L}}] \Vdash a\,[\mathbb{L}']) \end{cases}$$

C Types

We now describe typing. To shorten some definitions, we define a category of *identifiers*, which include bound names and atomic principals.

Syntax

$\eta \ ::=\ x \mid p \mid a \mid \alpha$	Identifiers
$\Delta \ ::=\ \cdot \mid \Delta, \eta : \mathscr{T} \mid \Delta, \phi \mid \Delta, V = M$	Environments
$\Phi \ ::=\ \cdot \mid \Phi, \phi \mid \Phi, V = M$	Logic Environments

Environments have two types of data: type bindings for names (as usual) and logical phrases, including equalities and predicates.

In our initial presentation, we will not be specific about the form of the logics. Specific requirements are given before the theorems, below, and we sketch an example logic afterwords.

In addition to the usual notion of values (i.e., no further reductions possible), the type system also formalizes "purity" annotations: Pure terms are are guaranteed to converge to a value without mutating the heap. An example of a pure term that is not a value is $V.\mathsf{loc}$.

Pure terms are used to formalize well-formed types.

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

$$\frac{}{\Delta \vdash \mathsf{Unit}} \quad \frac{}{\Delta \vdash \mathsf{Prin}} \quad \frac{}{\Delta \vdash \mathsf{Object}\texttt{<}\cdot\texttt{>}} \quad \frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}:\vec{P}\texttt{>} \quad \Delta \vdash \vec{\phi} : \vec{P}}{\Delta \vdash c\texttt{<}\vec{\phi}\texttt{>}} \quad \frac{\Delta \vdash \vec{\mathscr{T}}}{\Delta \vdash \mathsf{Pred}(\vec{\mathscr{T}})}$$

Note that Proc is not well formed, and thus cannot appear in an environment. The key new case in the above table is that for classes. The purity condition in this definition ensures that all the free names in $\vec{\phi}$ that are not bound in Δ are (hereditarily) immutable.

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

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

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

Subtyping is reflexive and transitive. As usual, the declared inheritances give rise to subtyping, as does the implication of the effects for the same base class. Subtyping is preserved by substitutions for the logic variables. Define $dom(\Delta) = \{\eta \mid \eta : \mathscr{T} \in \Delta\}$.

Well Formed Environment $(\Delta;\Sigma \vdash \diamond)$

$$\frac{\begin{array}{l}\Delta \ni x:\mathscr{T} \text{ implies } \mathscr{T} = \mathsf{Pred} \text{ or } (\exists T)\ \mathscr{T} = T \text{ and } \Delta \vdash T \\ \Delta \ni p:\mathscr{T} \text{ implies } (\exists C)\ \mathscr{T} = C \text{ and } \Delta \vdash C \\ \Delta \ni a:\mathscr{T} \text{ implies } \mathscr{T} = \mathsf{Prin} \\ \Delta \ni \alpha:\mathscr{T} \text{ implies } (\exists P)\ \mathscr{T} = P \text{ and } \Delta \vdash P \\ \Delta \ni V{=}M \text{ implies } (\exists T)\ \Delta \vdash V : T \text{ and } \Delta;\Sigma \vDash_a M : T\ \mathsf{Pure} \\ \Delta \ni p:\mathscr{T} \text{ implies } (\exists H)\Sigma \ni H \text{ and } H = p:c\{\vec{f}{=}\vec{V}\} \\ \Sigma \ni H \text{ implies } \Delta;\Sigma \vDash_a H : \mathsf{Proc}\ \mathsf{Pure} \\ \text{each element in } dom(\Delta) \text{ appears exactly once}\end{array}}{\Delta;\Sigma \vdash \diamond}$$

The function *heap* takes a term and returns its collection of heap elements, Σ. The function *env* likewise returns the collections of declarations, Δ.

Env $(env(M) = \Delta)$

$env(\mathsf{let}\ x{=}N;\ M) = env(N)$

$env(N \Vdash M) = env(N), env(M)$
$env(b\,[M]\,) = env(M)$
$env(\,(\nu p : C)\, M) = p : C, env(M)$
$env(M) = \cdot$ Otherwise

Heap $(heap(M) = \Sigma)$

$heap(p : c\{\vec{f} = \vec{V}\}) = p : c\{\vec{f} = \vec{V}\}$
$heap(\mathsf{let}\; x = N;\, M) = heap(N)$
$heap(N \Vdash M) = heap(N), heap(M)$
$heap(a\,[M]\,) = heap(M)$
$heap(\,(\nu p : C)\, M) = heap(M)$
$heap(M) = \cdot$ Otherwise

Definition 3. *$\Sigma \Vdash \theta \Downarrow \phi$ is defined to mean $\Sigma \Vdash \theta \rightarrow^* \Sigma \Vdash \phi \not\rightarrow$.*

The function *clauses* function takes an environment Δ and returns a logic environment Φ. The key cases extract effects from a declaration. For example:

$$clauses_{a\,[p : c\{\vec{f} = \vec{V}\}]}(p : C) = a \text{ says } (effect(C))[\mathsf{this} := x]$$

As expected, the extracted effects are relativized with respect to the location of the object.

Clauses $(clauses_\Sigma(\Delta) = \Phi)$

$clauses_\Sigma(\,\cdot\,) = \cdot$
$clauses_\Sigma(\Delta, x : C) = clauses_\Sigma(\Delta), x.\mathsf{loc} \text{ says } (effect(C))[\mathsf{this} := x]$
$clauses_\Sigma(\Delta, p : C) = clauses_\Sigma(\Delta), V \text{ says } (effect(C))[\mathsf{this} := p]$ where $\Sigma \Vdash p.\mathsf{loc} \Downarrow V$
$clauses_\Sigma(\Delta, \eta : \mathscr{T}) = clauses_\Sigma(\Delta)$ $\mathscr{T}$ not a class type
$clauses_\Sigma(\Delta, V = M) = clauses_\Sigma(\Delta), V = W$ where $\Sigma \Vdash M \Downarrow W$
$clauses_\Sigma(\Delta, \phi) = clauses_\Sigma(\Delta), \phi$

Well Formed Values $(\Delta \vdash V : \mathscr{T}\,)$

$$\frac{}{\Delta \vdash \mathsf{unit} : \mathsf{Unit}} \qquad \frac{\Delta \ni x : T}{\Delta \vdash x : T} \qquad \frac{\Delta \ni p : T}{\Delta \vdash p : T} \qquad \frac{\Delta \ni a : \mathsf{Prin}}{\Delta \vdash a : \mathsf{Prin}}$$

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

Well Formed Terms $(\Delta; \Sigma \vDash_a M : \mathscr{T}\,\rho)$ $(\rho ::= \mathsf{Pure} \mid \mathsf{Impure})$

$$\frac{\Delta \vdash V : \mathscr{T}}{\Delta; \Sigma \vDash_a V : \mathscr{T}\,\rho}$$

$$\frac{\Delta \backslash p; \Sigma \vdash \diamond \quad \Delta \vdash p : c{<}\vec{\phi}{>} \quad fields(c) = \vec{\mu}\;\vec{T}\;\vec{f} \quad \Delta \vdash \vec{V} : \vec{T}' \quad \vdash \vec{T}' <: \vec{T}}{\Delta; \Sigma \vDash_a p : c\{\vec{f} = \vec{V}\} : \mathsf{Proc}\,\rho}$$

$$\frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : C \quad \mathit{fields}(C) = \vec{\mu}\ \vec{T}\ \vec{f} \quad \mu_i = \mathsf{final}}{\Delta;\Sigma \vdash_a V.f_i : T_i\ \rho} \qquad \frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : C}{\Delta;\Sigma \vdash_a V.\mathsf{loc} : \mathsf{Prin}\ \rho}$$

$$\frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : T \quad \Delta \vdash W : S \quad \Delta, V{=}W;\Sigma \vdash_a M : \mathscr{T}'\ \rho \quad \Delta;\Sigma \vdash_a N : \mathscr{T}\ \rho \quad \vdash \mathscr{T}' <: \mathscr{T}}{\Delta;\Sigma \vdash_a \mathsf{if}\ V{=}W\ \mathsf{then}\ M\ \mathsf{else}\ N : \mathscr{T}\ \rho}$$

$$\frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : T \quad \Delta \vdash W : S \quad \Delta, V{=}W;\Sigma \vdash_a M : \mathscr{T}\ \rho \quad \Delta;\Sigma \vdash_a N : \mathscr{T}'\ \rho \quad \vdash \mathscr{T}' <: \mathscr{T}}{\Delta;\Sigma \vdash_a \mathsf{if}\ V{=}W\ \mathsf{then}\ M\ \mathsf{else}\ N : \mathscr{T}\ \rho}$$

$$\frac{\begin{array}{l}\Delta;\Sigma \vdash_a N : T\ \rho \quad \Delta, \mathit{env}(N);\Sigma, \mathit{heap}(N) \vdash_a N' : T\ \mathsf{Pure} \\ \Delta, \mathit{env}(N), x{:}T, x{=}N';\Sigma, \mathit{heap}(N) \vdash_a M : \mathscr{T}\ \rho \quad \mathit{right}(N) = N'\end{array}}{\Delta;\Sigma \vdash_a \mathsf{let}\ x{=}N; M : \mathscr{T}\ \rho}$$

$$\frac{\Delta \vdash b : \mathsf{Prin} \quad \Delta;\Sigma \vdash_b M : \mathscr{T}\ \rho}{\Delta;\Sigma \vdash_a b[M] : \mathscr{T}\ \rho} \qquad \frac{\Delta, p{:}C;\Sigma \vdash_a M : \mathscr{T}\ \rho}{\Delta;\Sigma \vdash_a (\nu p{:}C)\, M : \mathscr{T}\ \rho}$$

$$\frac{\begin{array}{l}\Delta, \mathit{env}(M);\Sigma, \mathit{heap}(M) \vdash_a N : \mathscr{S}\ \rho \\ \Delta, \mathit{env}(N);\Sigma, \mathit{heap}(N) \vdash_a M : \mathscr{T}\ \rho \\ \mathit{fn}(N \Vert\!\lceil M) \subseteq \mathit{dom}(\Delta)\end{array}}{\Delta;\Sigma \vdash_a N \Vert\!\lceil M : \mathscr{T}\ \rho}$$

In the rule for discharging conditionals, a predicate is added into the environment. We will discuss well-formed predicates later. In the rule for let, the equations are added to the typing environment only if the term N is pure.

The rule for located terms causes the expected switch of principal in the type judgement. The rules for new scoped object references and heap objects is as expected.

The rule for concurrent composition reflects the ideas from conjoining specifications of concurrent systems [46] — each component can assume the information exposed by the other component. The accumulation of effects from parallel components will aid in discharging the proof obligations that will be discussed in the forthcoming constructor and expectation rules.

Well Formed Terms (continued)

$$\frac{\Delta;\Sigma \vdash_a N : T\ \mathsf{Impure} \quad \Delta, \mathit{env}(N), x{:}T;\Sigma, \mathit{heap}(N) \vdash_a M : \mathscr{T}\ \rho}{\Delta;\Sigma \vdash_a \mathsf{let}\ x{=}N; M : \mathscr{T}\ \mathsf{Impure}}$$

$$\frac{\begin{array}{l}\Delta;\Sigma \vdash \diamond \quad \Delta \vdash c{<}\vec{\phi}{>} \\ \mathit{fields}(c{<}\vec{\phi}{>}) = \vec{\mu}\ \vec{T}\ \vec{f} \quad \Delta \vdash \vec{V} : \vec{T}' \quad \vdash \vec{T}' <: \vec{T} \\ \mathit{effect}(c{<}\vec{\phi}{>}) = \theta \quad (\Sigma \Vert\!\lceil a\,[p{:}c{<}\vec{\phi}{>}\{\vec{V}\}]\,) \Vert\!\lceil \theta[\mathsf{this} := p] \Downarrow \psi \\ \mathit{clauses}_\Sigma(\Delta) \vDash a\ \mathsf{says}\ \psi \qquad\qquad p \notin \mathit{fn}(\theta)\end{array}}{\Delta;\Sigma \vdash_a \mathsf{new}\ c{<}\vec{\phi}{>}(\vec{V}) : c{<}\vec{\phi}{>}\ \mathsf{Impure}}$$

$$\frac{\begin{array}{l}\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : C \quad \mathit{body}(C.\ell) = {<}\vec{\beta}:\vec{Q}{>}S(\vec{T}) \\ \Delta \vdash \vec{\phi} : \vec{Q} \quad \Delta \vdash \vec{W} : \vec{T}' \quad \vdash \vec{T}' <: \vec{T}[\vec{\beta} := \vec{\phi}] \\ \Delta, x{:}S[\vec{\beta} := \vec{\phi}], b{:}\mathsf{Prin}, b{=}V.\mathsf{loc}, \mathit{Prov}(x,b,a);\Sigma \vdash_a M : \mathscr{T}\ \rho \quad b \notin \mathit{fn}(M, \mathscr{T})\end{array}}{\Delta;\Sigma \vdash_a \mathsf{let}\ x{=}V.\ell{<}\vec{\phi}{>}(\vec{W}); M : \mathscr{T}\ \mathsf{Impure}}$$

$$\frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : C \quad \mathit{fields}(C) = \vec{\mu}\ \vec{T}\ \vec{f}}{\Delta;\Sigma \vdash_a V.f_i : T_i\ \mathsf{Impure}}$$

$$\frac{\Delta;\Sigma \vdash \diamond \quad \Delta \vdash V : C \quad \mathit{fields}(C) = \vec{\mu}\;\vec{T}\;\vec{f} \quad \mu_i = \mathsf{mutable} \quad \Delta \vdash W : T' \quad \vdash T' <: T_i}{\Delta;\Sigma \vdash_{a} V.f_i := W : \mathsf{Unit\ Impure}}$$

The constructor rule is a key rule in our system. 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. $\Sigma \Vdash \theta \Downarrow \phi$ is defined as $\Sigma \Vdash \theta \rightarrow^{*} \Sigma \Vdash \phi \not\rightarrow$ — this evaluation is guaranteed to terminate, and establishes the required bindings, including those of the immutable fields of the newly constructed object, into the class predicate that has been extracted. The actual proof obligation established is in the form of the utterance of the principal at whom the object is located, so the effect carried by an object is really uttered by its loc. The statements that can be used to discharge this proof obligation are derived from the environment via $\mathit{clauses}_{\Sigma}(\Delta)$ that accumulates the benefits derivable from the objects declared in the environment and the equations accumulated in the environment via lets and conditionals.

The rule for "generic" methods is standard, apart from the substitution of concrete formulas for the logical variables being carried in the method definition. Similarly, field gets and sets are standard.

The rule for statements ensures that the statement being made at location A is equivalent to an utterance of A — a formal treatment of this point of authorization logics is available in the background section on authorization logics in the appendix.

The last rule for expectations is the second place where proof obligations are established in the system. The accumulation of statements in the environment via $\mathit{clauses}_{\Sigma}(\Delta)$ is as in the constructor case — the static expectation annotation itself specifies the proof obligation.

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

$$\frac{\begin{array}{l}\Delta, \vec{\alpha} : \vec{P} \vdash D, \vec{T} \qquad \Delta, \vec{\alpha} : \vec{P}, \mathsf{this} : c\texttt{<}\vec{\alpha}\texttt{>}; \cdot \vdash \theta : \mathsf{Pred\ Pure} \\ \mathit{fields}(D) = \vec{\mu}_D\;\vec{T}_D\;\vec{f}_D \qquad \vec{f}_D\ \mathsf{disjoint}\ \vec{f} \\ \Delta \vdash \vec{\mathscr{M}}\ \mathit{in}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \lhd D\end{array}}{\Delta \vdash \mathsf{class}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \lhd D\{\vec{\mu}\;\vec{T}\;\vec{f};\ \vec{\mathscr{M}}\}\,[\theta]}$$

$$\frac{\begin{array}{l}\Delta, \vec{\alpha} : \vec{P}, \vec{\beta} : \vec{Q} \vdash S, \vec{T} \\ \Delta, \vec{\alpha} : \vec{P}, \vec{\beta} : \vec{Q}, \vec{x} : \vec{T}, \mathsf{caller} : \mathsf{Prin}, \mathsf{this} : c\texttt{<}\vec{\alpha}\texttt{>}, a : \mathsf{Prin}, a = \mathsf{this.loc}, \mathit{Prov}(\vec{x}, \mathsf{caller}, a); \cdot \vdash_{a} M : S'\ \rho \\ \vdash S' <: S \hfill a \notin \mathit{fn}(M) \\ \vdash \texttt{<}\vec{\beta}\texttt{>}S(\vec{T})\ \mathit{can\ override}\ D.\ell\end{array}}{\Delta \vdash \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\ \ell(\vec{T}\ \vec{x})\{M\}\ \mathit{in}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \lhd D}$$

Note that 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. In typing the method body, one can use the logical variables of the class and the method declaration, as well as the provenance of the parameters, which is expressed using the predicate *Prov*. We write $\mathit{Prov}(\vec{x}, \mathsf{caller}, a)$ as shorthand for $\mathit{Prov}(x_1, \mathsf{caller}, a), \ldots, \mathit{Prov}(x_n, \mathsf{caller}, a)$.

Correspondingly, *Prov* also appears in the rule for typing method call to allow the caller to use the provenance of the return value.

Results. We identify the properties required of the logic safety. These properties broadly fall into the following categories. Firstly, the closure of inference under the structural properties of exchange and weakening (so the underlying logic has to be affine and commutative) and transitivity via cut[3]. Secondly, the equality predicate is substitutive and closed wrt reduction. Finally, conditions on opponents.

Let the principal name "**0**" represents the most trustworthy principal, and "**1**" represents the least. We say that a logic is *enforceable* if the following properties hold. In this definition, we use σ to stand for for substitutions of pure terms M for x.

1. $\phi \vDash \phi$.
2. If $\Phi \vDash \psi$ then $\Phi, \Phi' \vDash \psi$, for any Φ'.
3. If $\Phi, \Phi', \Phi' \vDash \psi$ then $\Phi, \Phi' \vDash \psi$.
4. If $\Phi, \Phi', \Phi'' \vDash \phi$ then $\Phi, \Phi'', \Phi' \vDash \phi$.
5. If $\Phi, \phi \vDash \psi$ and $\Phi \vDash \phi$ then $\Phi \vDash \psi$.
6. If $\Phi \vDash \psi$ then $\Phi\sigma \vDash \psi\sigma$, for any substitution σ from variables to values, or from atomic principals to atomic principals.
7. If $\Phi, V = V, \Phi' \vDash \psi$ then $\Phi, \Phi' \vDash \psi$.
8. If $\Phi, \eta = V, \Phi' \vDash \psi$ then $\Phi, \Phi' \vDash \psi[\eta := V]$.
9. $\Phi \vDash \mathbf{1}$ says ψ, for any Φ, ψ.

An *opponent* is any process located at the principal **1**. From the final requirement, it follows that opponents may utter any clause and are thus completely free to construct any new objects.

Fix a class table $\vec{\mathscr{D}}$. The class table is well formed if $\vdash \mathscr{D}$, for every $\mathscr{D}$ in $\vec{\mathscr{D}}$. The concrete interpretation of the labelling functions and *tag* is enforceable if $\Delta \vdash V : T$ and $\Delta \vdash A : \mathsf{Prin}$ implies that $\Delta \vdash V : T$ and $\Delta \vdash \mathit{tag}(A, V) : T$.

The following results suppose that the class table is well formed, that the underlying logic is enforceable, and that the concrete interpretation of the labelling functions is enforceable.

Theorem 1 (Preservation). *Suppose that the class table is well formed with respect to Δ. If $\Delta; \cdot \vdash M$ and $M \to M'$ then $\Delta; \cdot \vdash M'$.*

Theorem 2 (Progress). *Suppose $\vec{p}:\vec{C};\Sigma \vdash M$. Then either right$(M)$ is a value, or $(\nu\vec{p}:\vec{C})\ \Sigma \Vdash M \to M'$ for some M'.*

Definition 4 (Safety). *A term M is* safe *if whenever $M \to^{*}\equiv (\nu\vec{p}:\vec{C})\ a[\mathsf{new}\ c(\vec{\phi})] \Vdash M'$, either $a = \mathbf{1}$ or clauses$_{heap(M')}(\vec{p}:\vec{C}, env(M')) \vDash effect(c{<}\vec{\phi}{>})$.*

Corollary 1 (Safety). *Suppose that $\vec{p}:\vec{C};\Sigma \vdash M$. Then $(\nu\vec{p}:\vec{C})\ \mathbf{1}[N] \Vdash \Sigma \Vdash a[M]$ is safe for any N such that $\vec{p}:\vec{C};\Sigma \vdash \mathbf{1}[N]$.*

[3] The type system does not require other logical connectives such as conjunction. If these were present in the logic, their normal properties would have to be enforced by their usual rules.

Safety requires that any objects created by trustworthy processes have their effects justified by the accumulated effects. The effects of objects created by opponent processes are not required to hold.

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

D Proofs

In all proofs we assume that the underlying logic is enforceable. We also assume that the class table $\vec{\mathscr{D}}$ is well formed.

Lemma 1 (Weakening). *Suppose* $\Delta;\Sigma \vDash_a M : \mathscr{T}\ \mathsf{Impure}$. *Then* $\Delta,\Delta';\Sigma,\Sigma' \vDash_a M : \mathscr{T}\ \mathsf{Impure}$ *if* $\Delta,\Delta';\Sigma,\Sigma' \vdash \diamond$.

Proof. *Follows from the standard argument and property 2 of the logic.*

Lemma 2 (Exchange). *Suppose* $\Delta,\Delta',\Delta'';\Sigma,\Sigma',\Sigma'' \vDash_a M : \mathscr{T}\ \mathsf{Impure}$, *and* $\mathit{fn}(\Delta'') \subseteq \mathit{dom}(\Delta)$. *Then* $\Delta,\Delta'',\Delta';\Sigma,\Sigma'',\Sigma' \vDash_a M : \mathscr{T}\ \mathsf{Impure}$.

Proof. *Follows from the standard argument and property 4 of the logic.*

Lemma 3 (Bounds Weakening). *Suppose* $\Delta,x:S;\Sigma \vDash_a M : T\ \rho$ *and* $\vdash S' <: S$. *Then* $\Delta,x:S';\Sigma \vDash_a M : T'\ \rho$ *where* $\vdash T' <: T$.

Proof. *By induction on the derivation of* $\Delta,x:S;\Sigma \vDash_a M : T\ \rho$.

Lemma 4 (Structural Equivalence Preservation by Substitution). *Suppose* $M \equiv N$. *Then* $M[x:=V] \equiv N[x:=V]$.

Proof. *By induction on the derivation of* $M \equiv N$.

Lemma 5 (Type Preservation by Substitution of Locations). *Suppose* $\Delta,b:\mathsf{Prin},\Delta';\Sigma \vDash_a M : \mathscr{T}\ \rho$ *and* $\Delta \vdash b' : \mathsf{Prin}$, *where* $b,b' \notin \mathit{fn}(M,\mathscr{T})$. *Then* $\Delta,\Delta'[b:=b'];\Sigma \vDash_{a[b:=b']} M : \mathscr{T}\ \rho$.

Proof. *By induction on the derivation of* $\Delta,b:\mathsf{Prin},\Delta';\Sigma \vDash_a M : \mathscr{T}\ \rho$.

Lemma 6 (Well-Formed Type Preservation by Substitution). *Suppose* $\Delta,x:T,\Delta' \vdash \mathscr{T}$ *and* $\Delta \vdash V : T$. *Then* $\Delta,\Delta'[x:=V] \vdash \mathscr{T}[x:=V]$.

Proof. *By induction on the derivation of* $\Delta,x:T,\Delta' \vdash \mathscr{T}$.

Lemma 7 (Well-Formed Environment Preservation by Substitution). *Suppose* $\Delta,x:T,\Delta';\Sigma \vdash \diamond$ *and* $\Delta \vdash V : T$. *Then* $\Delta,\Delta'[x:=V];\Sigma \vdash \diamond$.

Proof. *By induction on the derivation of* $\Delta,x:T,\Delta';\Sigma \vdash \diamond$, *appealing to Lemma 6.*

Lemma 8 (Subtype Preservation by Substitution). *Suppose* $\vdash T' <: T$. *Then* $\vdash T'[x:=V] <: T[x:=V]$ *for any* x,V.

Proof. *By induction on the derivation of* $\vdash T <: T'$, *appealing to property 6 of the logic.*

Lemma 9. *Suppose* $body(C.\ell) = \text{<}\vec{\beta} : \vec{Q}\text{>}S\,(\vec{T}\ \vec{x})\,\{M\}$*. Then* $body(C[x := V].\ell) = (\text{<}\vec{\beta} : \vec{Q}\text{>}S\,(\vec{T}\ \vec{x})\,\{M\})[x := V]$.

Proof. *Follows directly from the definition of body.*

Lemma 10. *Suppose* $body(C.\ell) = \text{<}\vec{\beta} : \vec{Q}\text{>}S\,(\vec{T}\ \vec{x})\,\{M\}$*. Then* $\Delta, \vec{\beta} : \vec{Q}, \mathsf{caller} : \mathsf{Prin}, \mathsf{this} : C, \vec{x} : \vec{T}; \Sigma \models_a M : S$ Impure *for any* Δ, Σ, a *where* $\Delta; \Sigma \vdash \diamond$ *and* $a \notin fn(M)$.

Lemma 11. *Suppose* $\Delta, env(N); \Sigma, heap(N) \models_a M : \mathscr{T}$ Impure *and* $N \rightarrow N'$ *for some* $M, N, a, \mathscr{T}$*. Then* $\Delta, env(N'); \Sigma, heap(N') \models_a M : \mathscr{T}$ Impure.

Proof. *By induction on the derivation of* $N \rightarrow N'$*. All cases are easy, appealing to weakening.*

Lemma 12. *Suppose* $\Delta, V = V; \Sigma \models_a M : \mathscr{T}$ Impure*. Then* $\Delta; \Sigma \models_a M : \mathscr{T}$ Impure.

Proof. *By induction on the derivation of* $\Delta, x : T, x = right(N); \Sigma \models_a M : \mathscr{T}$ Impure*. The only case that is affected is the case for new, which follows directly from property 8 of the logic.*

Lemma 13. *Suppose* $\Delta, V = N, \Delta'; \Sigma \models_{\overline{b}} M : \mathscr{T}\ \rho$*, and* $\Sigma \Vdash N \rightarrow \Sigma \Vdash N'$*, for some* N'*. Then* $\Delta, V = N', \Delta'; \Sigma \models_{\overline{b}} M : \mathscr{T}\ \rho$.

Proof. *By induction on the derivation of* $\Delta, V = N, \Delta'; \Sigma \models_{\overline{b}} M : \mathscr{T}\ \rho$*. The only case that is affected is the case for new, which follows from the basic properties of convergence.*

Lemma 14. *Suppose* $\Delta, x : T, x = right(N); \Sigma \models_a M : \mathscr{T}\ \rho$*, and* $\Delta; \Sigma \models_a N : T$ Pure*, and* $N \rightarrow N'$ *for some* N'*. Then* $\Delta, x : T, x = right(N'); \Sigma \models_a M : \mathscr{T}\ \rho$.

Proof. *By induction on the derivation of* $\Delta, x : T, x = right(N); \Sigma \models_a M : \mathscr{T}\ \rho$*. The only case that is affected is the case for new, which follows from the basic properties of convergence.*

Theorem 3 (Type Preservation by Structural Equivalence). *Suppose* $\Delta; \Sigma \models_{\overline{b}} M : \mathscr{T}\ \rho$ *and* $M \equiv M'$*. Then* $\Delta; \Sigma \models_{\overline{b}} M' : \mathscr{T}\ \rho$.

Proof. *By induction on the derivation of* $M \equiv M'$.

Case $(M \Vdash N) \Vdash L \equiv M \Vdash (N \Vdash L)$

($\longrightarrow$)
Assume $\Delta; \Sigma \models_{\overline{b}} (M \Vdash N) \Vdash L : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(L); \Sigma, heap(L) \models_{\overline{b}} M \Vdash N : \mathscr{T}'\ \rho$,
and, $\Delta, env(M), env(N); \Sigma, heap(M), heap(N) \models_{\overline{b}} L : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(L), env(N); \Sigma, heap(L), heap(N) \models_{\overline{b}} M : \mathscr{T}''\ \rho$,
and, $\Delta, env(L), env(M); \Sigma, heap(L), heap(M) \models_{\overline{b}} N : \mathscr{T}'\ \rho$.
By Lemma 2, $\Delta, env(M), env(L); \Sigma, heap(M), heap(L) \models_{\overline{b}} N : \mathscr{T}'\ \rho$.
By the type rule, $\Delta, env(M); \Sigma, heap(M) \models_{\overline{b}} N \Vdash L : \mathscr{T}\ \rho$.
By the type rule, $\Delta; \Sigma \models_{\overline{b}} M \Vdash (N \Vdash L) : \mathscr{T}\ \rho$.
($\longleftarrow$)
Similar argument.

Case $(M \mathbin{\|\!\lceil} N) \mathbin{\|\!\lceil} L \equiv (N \mathbin{\|\!\lceil} M) \mathbin{\|\!\lceil} L$

The left and right cases are symmetric.
Assume $\Delta;\Sigma \vdash_{b} (M \mathbin{\|\!\lceil} N) \mathbin{\|\!\lceil} L : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} M \mathbin{\|\!\lceil} N : \mathscr{T}'\ \rho$,
  and, $\Delta, env(M), env(N);\Sigma, heap(M), heap(N) \vdash_{b} L : \mathscr{T}\ \rho$.
By Lemma 2, $\Delta, env(N), env(M);\Sigma, heap(N), heap(M) \vdash_{b} L : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(L), env(N);\Sigma, heap(L), heap(N) \vdash_{b} M : \mathscr{T}''\ \rho$,
  and, $\Delta, env(L), env(M);\Sigma, heap(L), heap(M) \vdash_{b} N : \mathscr{T}'\ \rho$.
By the type rule, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} N \mathbin{\|\!\lceil} M : \mathscr{T}''\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{b} (N \mathbin{\|\!\lceil} M) \mathbin{\|\!\lceil} L : \mathscr{T}\ \rho$.

Case $M \mathbin{\|\!\lceil} ((\nu p{:}C)\, N) \equiv (\nu p)(M \mathbin{\|\!\lceil} N)$

(⟶)
Assume $\Delta;\Sigma \vdash_{b} M \mathbin{\|\!\lceil} ((\nu p)\, N) : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C, env(N);\Sigma, heap(N) \vdash_{b} M : \mathscr{T}'\ \rho$,
  and, $\Delta, env(M);\Sigma, heap(M) \vdash_{b} (\nu p{:}C)\, N : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(M), p{:}C;\Sigma, heap(M) \vdash_{b} N : \mathscr{T}\ \rho$.
By Lemma 2, $\Delta, p{:}C, env(M);\Sigma, heap(M) \vdash_{b} N : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C;\Sigma \vdash_{b} M \mathbin{\|\!\lceil} N : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{b} (\nu p{:}C)\, M \mathbin{\|\!\lceil} N : \mathscr{T}\ \rho$.
(⟵)
Similar argument, in reverse.

Case $\mathsf{let}\ x{=}(L \mathbin{\|\!\lceil} N);\, M \equiv L \mathbin{\|\!\lceil} (\mathsf{let}\ x{=}N;\, M)$

(⟶)
There are two matching type rules; we consider each separately.
(i) Assume $\Delta;\Sigma \vdash_{b} \mathsf{let}\ x{=}(L \mathbin{\|\!\lceil} N);\, M : \mathscr{T}\ \rho$ *by the first rule.*
By the type rule, $\Delta;\Sigma \vdash_{b} L \mathbin{\|\!\lceil} N : T'\ \rho$,
  and, $\Delta, env(N);\Sigma, heap(N) \vdash_{b} N' : T'\ \mathsf{Pure}$,
  and, $\vdash T' <: T$,
  and, $right(L \mathbin{\|\!\lceil} N) = N'$,
  and, $\Delta, env(L), env(N), x{:}T, x{=}N';\Sigma, heap(L), heap(N) \vdash_{b} M : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta, env(N) \vdash L : \mathscr{T}'\ \rho$,
  and, $\Delta, env(L) \vdash N : T\ \rho$.
By the type rule, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} \mathsf{let}\ x{=}N;\, M : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{b} L \mathbin{\|\!\lceil} (\mathsf{let}\ x{=}N;\, M) : \mathscr{T}\ \rho$.
(ii) Assume $\Delta;\Sigma \vdash_{b} \mathsf{let}\ x{=}(L \mathbin{\|\!\lceil} N);\, M : \mathscr{T}\ \rho$ *by the second rule.*
By the type rule, $\Delta;\Sigma \vdash_{b} L \mathbin{\|\!\lceil} N : T\ \rho$,
  and, $\Delta, env(L), env(N), x{:}T;\Sigma, heap(L), heap(N) \vdash_{b} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(N);\Sigma, heap(N) \vdash_{b} L : \mathscr{T}'\ \rho$,
  and, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} N : T\ \rho$.
By the type rule, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} \mathsf{let}\ x{=}N;\, M : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{b} L \mathbin{\|\!\lceil} : \mathsf{let}\ x{=}N;\, M\ \rho\,\mathscr{T}$.
(⟵)
Assume $\Delta;\Sigma \vdash_{b} L \mathbin{\|\!\lceil} (\mathsf{let}\ x{=}N;\, M) : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(N);\Sigma, heap(N) \vdash_{b} L : \mathscr{T}'\ \rho$,
  and, $\Delta, env(L);\Sigma, heap(L) \vdash_{b} \mathsf{let}\ x{=}N;\, M : \mathscr{T}\ \rho$.
There are two matching type rules; we consider each separately.

(i) Assume $\Delta, env(L); \Sigma, heap(L) \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$ *by the first.*
By the type rule, $\Delta, env(L); \Sigma, heap(L) \Vdash_{b} N : T'\ \rho$,
  and, $\Delta, env(L); \Sigma, heap(L) \Vdash_{b} N' : T'\ \mathsf{Pure}$,
  and, $\vdash T' <: T$,
  and, $\Delta, env(L), env(N), x{:}T, x{=}N'; \Sigma, heap(L), heap(N) \Vdash_{b} M : \mathscr{T}\ \mathsf{Pure}$,
  and, $right(N) = N'$.
By the type rule, $\Delta; \Sigma \Vdash_{b} (L \mathbin{\|\!\lceil} N) : T'\ \rho$.
From def. of right, it is easy to see that $right(L \mathbin{\|\!\lceil} N) = N'$.
By the type rule, $\Delta; \Sigma \Vdash_{b} \mathsf{let}\ x{=}(L \mathbin{\|\!\lceil} N);\ M : \mathscr{T}\ \rho$.
(ii) Assume $\Delta, env(L); \Sigma, heap(L) \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$ *by the second.*
By the type rule, $\Delta, env(L); \Sigma, heap(L) \Vdash_{b} N : T'\ \rho$,
  and $\vdash T' <: T$,
  and $\Delta, env(L), env(N), x{:}T; \Sigma, heap(L), heap(N) \Vdash_{b} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta; \Sigma \Vdash_{b} N : T'\ \rho$.
By the type rule, $\Delta; \Sigma \Vdash_{b} \mathsf{let}\ T = (L \mathbin{\|\!\lceil} N);\ M : \mathscr{T}\ \rho$.

Case $\mathsf{let}\ x{=}(\,(\nu p{:}C)\ N);\ M \equiv (\nu p{:}C)(\mathsf{let}\ x{=}N;\ M)$
By hypothesis, $p \notin fn(M)$.
$(\longrightarrow)$
There are two matching type rules; we consider each separately.
(i) Assume $\Delta; \Sigma \Vdash_{b} \mathsf{let}\ x{=}(\,(\nu p{:}C)\ N);\ M : \mathscr{T}\ \rho$ *by the first rule.*
By the type rule, $\Delta; \Sigma \Vdash_{b} (\nu p{:}C)\ N : T'\ \rho$,
  and $\Delta, p{:}C, env(N); \Sigma, heap(N) \Vdash_{b} N' : T'\ \mathsf{Pure}$,
  and $\vdash T' <: T$,
  and $right(\,(\nu p{:}C)\ N) = N'$,
  and $\Delta, p{:}C, env(N), x{:}T, x{=}N'; \Sigma, heap(N) \Vdash_{b} M : \mathscr{T}\ \rho$.
By def. $right(N) = N'$.
By the type rule, $\Delta, p{:}C; \Sigma \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$.
By the type rule, $\Delta; \Sigma \Vdash_{b} (\nu p{:}C)\ (\mathsf{let}\ x{=}N;\ M) : \mathscr{T}\ \rho$.
(ii) Assume $\Delta; \Sigma \Vdash_{b} \mathsf{let}\ x{=}(\,(\nu p{:}C)\ N);\ M : \mathscr{T}\ \rho$ *by the second rule.*
By the type rule, $\Delta; \Sigma \Vdash_{b} (\nu p{:}C)\ N : T'\ \rho$,
  and $\vdash T' <: T$,
  and, $\Delta, p{:}C, env(N), x{:}T; \Sigma, heap(N) \Vdash_{b} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C; \Sigma \Vdash_{b} N : T'\ \rho$.
By the type rule, $\Delta, p{:}C; \Sigma \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$.
By the type rule, $\Delta; \Sigma \Vdash_{b} (\nu p{:}C)\ (\mathsf{let}\ x{=}N;\ M) : \mathscr{T}\ \rho$.
$(\longleftarrow)$
Assume $\Delta; \Sigma \Vdash_{b} (\nu p{:}C)(\mathsf{let}\ x{=}N;\ M) : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C; \Sigma \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$.
There are two matching type rules; we consider each separately.
(i) Assume $\Delta, p{:}C; \Sigma \Vdash_{b} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho$ *by the first rule.*
By the type rule, $\Delta, p{:}C; \Sigma \Vdash_{b} N : T'\ \rho$,
  and, $\Delta, p{:}C, env(N); \Sigma, heap(N) \Vdash_{b} N' : T'\ \mathsf{Pure}$,
  and, $\vdash T' <: T$,
  and, $\Delta, p{:}C, env(N), x{:}T, x{=}N'; \Sigma, heap(N) \Vdash_{b} M : \mathscr{T}\ \rho$,
  and, $right(N) = N'$.

By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} (\nu p{:}C)\, N : T'\ \rho$.
From the def. $right(\,(\nu p{:}C)\, N) = N'$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} \mathsf{let}\ x{=}(\,(\nu p{:}C)\, N); M : \mathscr{T}\ \rho$.
(ii) Assume $\Delta, p{:}C;\Sigma \vdash_{\overline{b}} \mathsf{let}\ x{=}N; M : \mathscr{T}\ \rho$ *by the second rule.*
By the type rule, $\Delta, p{:}C;\Sigma \vdash_{\overline{b}} N : T'\ \rho$,
  and, $\vdash T' <: T$,
  and, $\Delta, p{:}C, env(N), x{:}T;\Sigma, heap(N) \vdash_{\overline{b}} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} (\nu p{:}C)\, N : T'\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} \mathsf{let}\ x{=}(\,(\nu p{:}C)\, N); M : \mathscr{T}\ \rho$.

Case $a\,[V] \equiv V$
(⟶)
Assume $\Delta;\Sigma \vdash_{\overline{b}} a\,[V] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a : \mathsf{Prin}$,
  and, $\Delta;\Sigma \vdash_{\overline{a}} V : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash V : \mathscr{T}$.
By the type rule, $\Delta \vdash V : \mathscr{T}\ \mathsf{Pure}$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} V : \mathscr{T}\ \rho$.
(⟵)
Immediate from type rule.

Case $a\,[N \parallel\!\lceil M] \equiv a\,[N] \parallel\!\lceil a\,[M]$
(⟶)
Assume $\Sigma;\Delta \vdash_{\overline{b}} a\,[N \parallel\!\lceil M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a : \mathsf{Prin}$,
  and, $\Delta;\Sigma \vdash_{\overline{a}} N \parallel\!\lceil M : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(M);\Sigma, heap(M) \vdash_{\overline{a}} N : \mathscr{T}'\ \rho$,
  and, $\Delta, env(N);\Sigma, heap(N) \vdash_{\overline{a}} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta, env(M);\Sigma, heap(M) \vdash_{\overline{b}} a\,[N] : \mathscr{T}'\ \rho$,
  and, $\Delta, env(N);\Sigma, heap(N) \vdash_{\overline{b}} a\,[M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a\,[N] \parallel\!\lceil a\,[M] : \mathscr{T}\ \rho$.
(⟵)
Same argument in reverse.

Case $a\,[\,(\nu p{:}C)\, N] \equiv (\nu p{:}C)\, a\,[N]$
(⟶)
Assume $\Delta;\Sigma \vdash_{\overline{b}} a\,[\,(\nu p{:}C)\, N] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a : \mathsf{Prin}$,
  and, $\Delta;\Sigma \vdash_{\overline{a}} (\nu p{:}C)\, N : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C;\Sigma \vdash_{\overline{a}} N : \mathscr{T}\ \rho$.
By the type rule, $\Delta, p{:}C;\Sigma \vdash_{\overline{b}} a\,[N] : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} (\nu p{:}C)\, a\,[N] : \mathscr{T}\ \rho$.
(⟵)
Same argument in reverse.

Case $a\,[\mathsf{let}\ x{=}N; M] \equiv \mathsf{let}\ x{=}a\,[N]\,; a\,[M]$
(⟶)
Assume $\Delta;\Sigma \vdash_{\overline{b}} a\,[\mathsf{let}\ x{=}N; M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a : \mathsf{Prin}$ *, and,* $\Delta;\Sigma \vdash_{\overline{a}} \mathsf{let}\ x{=}N; M : \mathscr{T}\ \mathsf{Pure}$.
For the latter, there are two type rules that match.

If the first, then $\Delta;\Sigma \vdash_{\overline{b}} N : T'\ \rho$,
and, $\Delta, env(N);\Sigma, heap(N) \vdash_{\overline{b}} N' : T'\ \mathsf{Pure}$,
and, $\vdash T' <: T$,
and, $\Delta, x : T, x = N';\Sigma \vdash_{\overline{a}} M : \mathscr{T}\ \rho$, *where* $N' = right(N)$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a\,[N] : T'\ \rho$,
and, $\Delta, x : T, x = N;\Sigma \vdash_{\overline{b}} a\,[M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} \mathsf{let}\ x = a\,[N]\,;\,a\,[M] : \mathscr{T}\ \rho$.
If the second, then $\Delta;\Sigma \vdash_{\overline{a}} N : T'\ \rho$,
and, $\vdash T' <: T$,
and, $\Delta, env(N), x : T;\Sigma, heap(M) \vdash_{\overline{a}} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a\,[N] : T\ \rho$,
and, $\Delta, env(N) x : T;\Sigma, heap(M) \vdash_{\overline{b}} a\,[M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} \mathsf{let}\ x = a\,[N]\,;\,a\,[M] : \mathscr{T}\ \rho$.
($\longleftarrow$*)*
Essentially the same argument in reverse.

Case $a_1\,[a_2\,[M]\,] \equiv a_2\,[M]$
($\longrightarrow$*)*
Assume $\Delta;\Sigma \vdash_{\overline{b}} a_1\,[a_2\,[M]\,] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a_1 : \mathsf{Prin}$,
and, $\Delta;\Sigma \vdash_{\overline{a_1}} a_2\,[M] : \mathscr{T}\ \rho$.
By the type rule, $\Delta \vdash a_2 : \mathsf{Prin}$,
and, $\Delta;\Sigma \vdash_{\overline{a_2}} M : \mathscr{T}\ \rho$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a_2\,[M] : \mathscr{T}\ \rho$.
($\longleftarrow$*)*
Direct from type rule. □

Lemma 15 (Type Preservation by Substitution into Values). *Suppose* $\Delta, x : T, \Delta' \vdash W : \mathscr{T}$ *, and* $\Delta \vdash V : T'$ *, and* $\vdash T' <: T$*. Then* $\Delta, \Delta'[x := V] \vdash W[x := V] : \mathscr{T}'$ *where* $\vdash \mathscr{T}' <: \mathscr{T}[x := V]$.

Proof. *By induction on the derivation of* $\Delta, x : T, \Delta' \vdash W : \mathscr{T}$ *. All cases are easy, we show one as an example.*

Case $\phi\,(\vec{W})$**:**
Assume $\Delta, x : T, \Delta' \vdash \phi\,(\vec{W}) : \mathscr{T}$.
By the type rule, $\Delta, x : T, \Delta' \vdash \phi : \mathsf{Pred}\,(\vec{\mathscr{T}})$,
and, $\Delta, x : T, \Delta' \vdash \vec{W} : \vec{\mathscr{T}}$.
By IH, $\Delta, \Delta'[x := V] \vdash \phi[x := V] : \mathsf{Pred}\,(\vec{\mathscr{T}})\,[x := V]$,
and, $\Delta, \Delta'[x := V] \vdash \vec{W}[x := V] : \vec{\mathscr{T}}[x := V]$.
By the type rule, $\Delta, \Delta'[x := V] \vdash \phi[x := V]\,(\vec{W}[x := V]) : \mathsf{Pred}$.

Lemma 16 (Pure Type Preservation by Substitution). *Suppose* $\Delta, x : T, \Delta';\Sigma \vdash_{\overline{a}} M : \mathscr{T}\ \mathsf{Pure}$*, and* $\Delta \vdash V : T'$ *, and* $\vdash T' <: T$*. Then* $\Delta, \Delta'[x := V];\Sigma \vdash_{\overline{a}} M[x := V] : \mathscr{T}'\ \mathsf{Pure}$ *where* $\vdash \mathscr{T}' <: \mathscr{T}[x := V]$.

Proof. *By induction on the derivation of* $\Delta, x : T, \Delta';\Sigma \vdash_{\overline{a}} M : \mathscr{T}\ \mathsf{Pure}$*. All cases are easy, we show one as an example.*

Case $p : c\{\vec{\boldsymbol{f}} = \vec{W}\}$**:**

Assume $\Delta, x : T, \Delta'; \Sigma \vDash_a p : c\{\vec{f} = \vec{W}\} : \mathsf{Proc\ Pure}$, *and* $\Delta \vdash V : T'$, *and* $\vdash T' <: T$.
By the type rule, $\Delta, x : T, \Delta' \vdash p : c\texttt{<}\vec{\phi}\texttt{>}$,
and, fields$(c) = \vec{\mu}\ \vec{S}\ \vec{f}$,
and, $\Delta, x : T, \Delta' \vdash \vec{W} : \vec{S}'$,
and, $\vdash \vec{S}' <: \vec{S}$.
By lemma 15, $\Delta, \Delta'[x := V] \vdash p : S''$ *where* $\vdash S'' <: c\texttt{<}\vec{\phi}\texttt{>}$.
By the type rules and def. of well formed env, $S'' = c\texttt{<}\vec{\phi}\texttt{>}$.
By lemma 15, $\Delta, \Delta'[x := V] \vdash \vec{W}[x := V] : \vec{S}''$ *where* $\vdash \vec{S}'' <: \vec{S}'$.
By the transitivity of subtyping, $\vdash \vec{S}'' <: \vec{S}$.
By the type rule, $\Delta, \Delta'[x := V] \vdash p : c\{\vec{f} = \vec{W}[x := V]\} : \mathsf{Proc\ Pure}$. □

Lemma 17 (Pure Type Preservation by Evaluation). *Suppose* $\Delta; \Sigma \vDash_b M : \mathscr{T}\ \mathsf{Pure}$ *and* $M \to M'$. *Then* $\Delta; \Sigma \vDash_b M' : \mathscr{T}'\ \mathsf{Pure}$ *where* $\vdash \mathscr{T}' <: \mathscr{T}$.

Proof. *By induction on the derivation of* $M \to M'$.

Case $\begin{pmatrix} a\,[p : c\{\vec{\boldsymbol{f}} = \vec{V}\}] \parallel \\ p.\boldsymbol{f}_i \end{pmatrix} \to \begin{pmatrix} a\,[p : c\{\vec{\boldsymbol{f}} = \vec{V}\}] \parallel \\ V_i \end{pmatrix}$**:**

Assume $\Delta; \Sigma \vDash_b a\,[p : c\{\vec{f} = \vec{V}\}] \parallel p.f_i : \mathscr{T}\ \mathsf{Pure}$.
By the type rule, $\mathscr{T}$ *is of the form* T,
and, $\Delta; \Sigma \vDash_b a\,[p : c\{\vec{f} = \vec{V}\}] : \mathscr{T}'\ \mathsf{Pure}$,
and, $\Delta; \Sigma, a\,[p : c\{\vec{f} = \vec{V}\}] \vDash_b p.f_i : \mathscr{T}\ \mathsf{Pure}$,
and, fn$(a\,[p : c\{\vec{f} = \vec{V}\}] \parallel p.f_i) \subseteq dom(\Delta)$.
By the type rule, $\Delta \vdash p : c\{\vec{f} = \vec{V}\} : \mathscr{T}'\ \mathsf{Pure}$.
By the type rule, $\Delta \vdash p : c\texttt{<}\vec{\phi}\texttt{>}$,
and, fields$(c) = \vec{\mu}\ \vec{T}\ \vec{f}$,
and, $\Delta \vdash V_i : T_i'$,
and, $\vdash T_i' <: T_i$.
By the type rule, $\Delta \vdash V_i : T_i'\ \mathsf{Pure}$.
By the type rule, $\Delta \vdash p : C$,
and, fields$(C) = \vec{\mu}''\ \vec{T}''\ \vec{f}''$,
and, $\mu_i' = \mathsf{final}$.
It is easy to see that $C = c\texttt{<}\vec{\phi}\texttt{>}$,
therefore $\mu_i = \mathsf{final}$ *and* $\mathscr{T} = T_i'$.

Case $\begin{pmatrix} a\,[p : c\{\vec{\boldsymbol{f}} = \vec{V}\}] \parallel \\ p.\mathsf{loc} \end{pmatrix} \to \begin{pmatrix} a\,[p : c\{\vec{\boldsymbol{f}} = \vec{V}\}] \parallel \\ a \end{pmatrix}$**:**

Assume $\Delta; \Sigma \vDash_b a\,[p : c\{\vec{f} = \vec{V}\}] \parallel p.\mathsf{loc} : \mathscr{T}\ \mathsf{Pure}$.
By def. of env, env$(a\,[p : c\{\vec{f} = \vec{V}\}]) = \cdot$,
and, env$(p.\mathsf{loc}) = \cdot$.
By the type rule, $\mathscr{T}$ *is of the form* T,
and, $\Delta; \Sigma \vDash_b a\,[p : c\{\vec{f} = \vec{V}\}] : \mathscr{T}'\ \mathsf{Pure}$,
and, $\Delta; \Sigma, a\,[p : c\{\vec{f} = \vec{V}\}] \vDash_b p.\mathsf{loc} : \mathscr{T}\ \mathsf{Pure}$,
and, fn$(a\,[p : c\{\vec{f} = \vec{V}\}] \parallel a_2\,[p.\mathsf{loc}]) \subseteq dom(\Delta)$.
By the type rule, $\mathscr{T} = \mathsf{Prin}$.
By the type rule, $\Delta \vdash a : \mathsf{Prin}$.

By the type rule, $\Delta;\Sigma \vdash_{\overline{a}_2} a : \mathscr{T}$ Pure.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a\,[p : c\{\vec{f} = \vec{V}\}] \Vert\!\ulcorner a : \mathscr{T}$ Pure.
Case if $V = V$ then M else $N \longrightarrow M$**:**
Assume $\Delta;\Sigma \vdash_{\overline{a}}$ if $V = V$ then M else $N : \mathscr{T}$ Pure.
By the type rule, $\Delta, V = V;\Sigma \vdash_{\overline{a}} M : \mathscr{T}$ Pure.
By lemma 12, $\Delta;\Sigma \vdash_{\overline{a}} M : \mathscr{T}$ Pure.
Case if $V = W$ then M else $N \longrightarrow N$**:**
By hypothesis, $V \neq W$.
Assume $\Delta;\Sigma \vdash_{\overline{a}}$ if $V = W$ then M else $N : \mathscr{T}$ Pure.
By the type rule, $\Delta;\Sigma \vdash_{\overline{a}} N : \mathscr{T}$ Pure.
Case let $x = V;\, M \longrightarrow M[x := V]$**:**
There are two matching type rules.
Assume $\Delta;\Sigma \vdash_{\overline{b}}$ let $x = V;\, M : \mathscr{T}$ Pure *by the first rule.*
By the type rule, $\Delta;\Sigma \vdash_{\overline{a}} V : T'$ Pure,
and, $\vdash T' <: T$,
and, $\Delta, x : T, x = V;\Sigma \vdash_{\overline{a}} M : \mathscr{T}$ Pure.
By Lemma 16, $\Delta, V = V;\Sigma \vdash_{\overline{a}} M[x := V] : \mathscr{T}'$ Pure, *where* $\vdash \mathscr{T}' <: \mathscr{T}$.
By Lemma 12, $\Delta;\Sigma \vdash_{\overline{a}} M[x := V] : \mathscr{T}'$ Pure.
Case $b\,[M] \longrightarrow b\,[M']$
By hypothesis, $M \longrightarrow M'$.
Assume $\Delta;\Sigma \vdash_{\overline{a}} b\,[M] : \mathscr{T}$ Pure.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} M : \mathscr{T}$ Pure.
By the IH, $\Delta;\Sigma \vdash_{\overline{b}} M' : \mathscr{T}'$ Pure, *where* $\vdash \mathscr{T}' <: \mathscr{T}$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{a}} b\,[M'] : \mathscr{T}'$ Pure.
Case let $x = N;\, M \longrightarrow$ let $x = N';\, M$
By hypothesis, $N \longrightarrow N'$.
Assume $\Delta;\Sigma \vdash_{\overline{b}}$ let $x = N;\, M : \mathscr{T}$ Pure.
By the type rule, $\Delta \vdash N : T'$ Pure,
and, $\vdash T' <: T$,
and, $\Delta, x : T, x = \mathit{right}(N);\Sigma \vdash_{\overline{b}} M : \mathscr{T}$ Pure.
By the IH, $\Delta \vdash N' : T'$ Pure.
By Lemma 14, $\Delta, x : T, x = \mathit{right}(N');\Sigma \vdash_{\overline{b}} M : \mathscr{T}$ Pure.
By type rule, $\Delta;\Sigma \vdash_{\overline{b}}$ let $x = N';\, M : \mathscr{T}$ Pure.
Case $M \longrightarrow M'$ **(where** $M \equiv N \longrightarrow N' \equiv M'$**)**
Follows easily from induction hypothesis and Theorem 3.
Case $M \Vert\!\ulcorner N \longrightarrow M' \Vert\!\ulcorner N$
By hypothesis, $M \longrightarrow M'$.
Assume that $\Delta;\Sigma \vdash_{\overline{a}} M \Vert\!\ulcorner N : \mathscr{T}$ Pure.
By the type rule, $\Delta, \mathit{env}(N);\Sigma, \mathit{heap}(N) \vdash_{\overline{a}} M : \mathscr{T}'$ Pure,
and, $\Delta, \mathit{env}(M);\Sigma, \mathit{heap}(M) \vdash_{\overline{a}} N : \mathscr{T}$ Pure.
By IH, $\Delta, \mathit{env}(N);\Sigma, \mathit{heap}(N) \vdash_{\overline{a}} M' : \mathscr{T}''$ Pure, *where* $\vdash \mathscr{T}'' <: \mathscr{T}'$.
By Lemma 11, $\Delta, \mathit{env}(M');\Sigma, \mathit{heap}(M') \vdash_{\overline{a}} N : \mathscr{T}$ Pure.
By the type rule, $\Delta;\Sigma \vdash_{\overline{a}} M' \Vert\!\ulcorner N : \mathscr{T}$ Pure.
Case $M \Vert\!\ulcorner N \longrightarrow M \Vert\!\ulcorner N'$
By hypothesis, $N \longrightarrow N'$.

Assume that $\Delta;\Sigma \vDash_{\overline{a}} M \Vert N : \mathscr{T}$ Pure.
By the type rule, $\Delta, env(N);\Sigma, heap(N) \vDash_{\overline{a}} M : \mathscr{T}''$ Pure,
and, $\Delta, env(M);\Sigma, heap(M) \vDash_{\overline{a}} N : \mathscr{T}$ Pure.
By IH, $\Delta, env(M) \vdash N' : \mathscr{T}'$ Pure, *where* $\vdash \mathscr{T}' <: \mathscr{T}$.
By Lemma 11, $\Delta, env(N') \vdash M : \mathscr{T}$ Pure.
By the type rule, $\Delta;\Sigma \vDash_{\overline{a}} M' \Vert N : \mathscr{T}$ Pure.
Case $(\nu p)\, M \to (\nu p)\, M'$
Follows easily from induction hypothesis. □

Lemma 18. *Suppose* $\Delta, x : T;\Sigma \vDash_{\overline{a}} M$: Pred Pure *and* $M \to M'$. *Then* $M[x := V] \to M'[x := V]$ *for any* x, V.

Proof. *By induction on the derivation of* $M \to M'$.

Case $\begin{pmatrix} b\,[p{:}c\{\boldsymbol{f} = V \cdots\}] \Vert \\ p.\boldsymbol{f} \end{pmatrix} \to \begin{pmatrix} b\,[p{:}c\{\boldsymbol{f} = V \cdots\}] \Vert \\ V \end{pmatrix}$**:**
Immediate.
Case let $y = V;\, M \to M[y := V]$**:**
Immediate.
Case let $y = N;\, M \to$ let $y = N';\, M$
By hypothesis, $N \to N'$.
Assume $\Delta, x : T;\Sigma \vDash_{\overline{b}}$ let $y = N;\, M : \mathscr{T}$ Pure, *and* $\Delta;\Sigma \vDash_{\overline{b}} V : T$ Pure.
By the type rule, $\Delta, x : T;\Sigma \vDash_{\overline{b}} N : S'$ Pure,
and, $\vdash S' <: S$,
and, $\Delta, x : T, y : S, y = right(N);\Sigma \vDash_{\overline{b}} M : \mathscr{T}$ Pure.
By Lemma 16, $\Delta;\Sigma \vDash_{\overline{b}} N[x := V] : S'[x := V]$ Pure.
By Lemma 8, $\vdash S'[x := V] <: S[x := V]$.
By IH, $N[x := V] \to N'[x := V]$.
By the evaluation rule, let $y = N[x := V];\, M[x := V] \to$ let $y = N'[x := V];\, M[x := V]$.
Case $M \to M'$**:**
By hypothesis, $M \equiv N \to N' \equiv M'$.
Assume $\Delta, x : T;\Sigma \vDash_{\overline{b}} M : \mathscr{T}$ Pure, *and* $\Delta;\Sigma \vDash_{\overline{b}} V : T$ Pure.
By Theorem 4, $\Delta;\Sigma \vDash_{\overline{b}} N[x := V] : \mathscr{T}$ Pure.
By IH, $N[x := V] \to N'[x := V]$.
By Lemma 4, $N'[x := V] \equiv M'[x := V]$.
By the evaluation rule, $M[x := V] \to M'[x := V]$.
Case $M \Vert N \to M' \Vert N$
By hypothesis, $M \to M'$.
Assume that $\Delta, x : T;\Sigma \vDash_{\overline{a}} M \Vert N : \mathscr{T}$ Pure.
By the type rule, $\Delta, x : T, env(N);\Sigma, heap(N) \vDash_{\overline{a}} M : \mathscr{T}'$ Pure,
and, $\Delta, x : T, env(M);\Sigma, heap(M) \vDash_{\overline{a}} N : \mathscr{T}$ Pure.
By IH, $M[x := V] \to M'[x := V]$.
By the evaluation rule, $M[x := V] \Vert N[x := V] \to M'[x := V] \Vert N[x := V]$.
Case $M \Vert N \to M \Vert N'$
By hypothesis, $N \to N'$.
Assume that $\Delta, x : T;\Sigma \vDash_{\overline{a}} M \Vert N : \mathscr{T}$ Pure.
By the type rule, $\Delta, x : T, env(N);\Sigma, heap(N) \vDash_{\overline{a}} M : \mathscr{T}'$ Pure,

and, $\Delta, env(M); \Sigma, heap(M) \vDash_a N : \mathscr{T}$ Pure.
By IH, $N[x := V] \rightarrow N'[x := V]$.
By the evaluation rule, $M[x := V] \Vdash N[x := V] \rightarrow M[x := V] \Vdash N'[x := V]$.

Case $(\nu p)\, M \rightarrow (\nu p)\, M'$
Follows directly from induction hypothesis.

Lemma 19. *Suppose effect*$(C) = \theta$ *and* $\Sigma \Vdash \theta \Downarrow \psi$. *Then* $\Sigma \Vdash \theta[x := V] \Downarrow \psi[x := V]$ *for any* C, x, V.

Proof. *A corollary of the previous lemma.*

Theorem 4 (Type Preservation by Substitution). *Suppose* $\Delta, x : T, \Delta'; \Sigma \vDash_a M : \mathscr{T}$ Impure *and* $\Delta \vdash V : T$. *Then* $\Delta, \Delta'[x := V]; \Sigma \vDash_a M[x := V] : \mathscr{T}'$ Impure, *where* $\vdash \mathscr{T}' <: \mathscr{T}[x := V]$.

Proof. *By induction on the derivation of* $\Delta, x : T, \Delta'; \Sigma \vDash_a M : \mathscr{T}$ Impure. *Cases involving values and pure terms are by appeal to theorems 15 and 16. Cases involving impure terms that have matching rules for pure terms follow the same logic, but with the trivial addition of a store. The remaining cases for impure terms are shown here.*

Case $\Delta, x : T, \Delta'; \Sigma \vDash_a \mathsf{new}\ c\text{<}\vec{\phi}\text{>}(\vec{W}) : c\text{<}\vec{\phi}\text{>}$ Impure**:**
Assume $\Delta \vdash V : T$.
By the type rule, $\Delta, x : T, \Delta'; \Sigma \vdash \diamond$,
and, $\Delta, x : T, \Delta' \vdash c\text{<}\vec{\phi}\text{>}$,
and, fields$(c\text{<}\vec{\phi}\text{>}) = \vec{\mu}\ \vec{T}\ \vec{f}$,
and, $\Delta, x : T, \Delta'; \Sigma \vDash_a \vec{W} : \vec{T}'$ Impure,
and, $\vdash \vec{T}' <: \vec{T}$,
and, effect$(c\text{<}\vec{\phi}\text{>}) = \theta$,
and, $(\Sigma \Vdash a\,[p : c\text{<}\vec{\phi}\text{>}\{\vec{V}\}]\,) \Vdash \theta[\mathsf{this} := p] \Downarrow \psi$,
and, clauses$(\Delta, x : T, \Delta') \vDash a$ *says* ψ.
By def. wfe., $\Delta, \Delta'[x := V]; \Sigma \vdash \diamond$.
By IH, $\Delta, \Delta'[x := V] \vdash c\text{<}\vec{\phi}\text{>}[x := V]$.
By def. of fields, fields$(c\text{<}\vec{\phi}\text{>}[x := V]) = \vec{\mu}\ \vec{T}\ \vec{f}[x := V]$,
By IH, $\Delta, \Delta'[x := V]; \Sigma \vDash_a \vec{W}[x := V] : \vec{T}'[x := V]$ Impure.
By Lemma 8, $\vdash \vec{T}'[x := V] <: \vec{T}[x := V]$.
From def. of effect,
it is easy to see that effect$(c\text{<}\vec{\phi}\text{>}[x := V]) = \theta[x := V]$.
By Lemma 19,
$(\Sigma \Vdash a\,[p : c\text{<}\vec{\phi}\text{>}\{\vec{V}\}]\,) \Vdash \theta[\mathsf{this} := p][x := V] \Downarrow \psi[x := V]$.
By def. of clauses,
clauses$(\Delta), x.\mathsf{loc}$ *says* $\theta[\mathsf{this} := x], \Delta' \vDash a$ *says* ψ.
By property 6 of the logic,
clauses$(\Delta), V.\mathsf{loc}$ *says* $\theta[\mathsf{this} := V], \Delta'[x := V]$
$\vDash a$ *says* $\psi[x := V]$.
By def. of clauses,
clauses$(\Delta, \Delta'[x := V]) \vDash a$ *says* $\psi[x := V]$.
By the type rule,
$\Delta, \Delta'[x := V]; \Sigma \vDash_a \mathsf{new}\ c\text{<}\vec{\phi}\text{>}(\vec{W})\,[x := V] : \mathsf{Proc}$ Impure.

Case $\mathsf{let}\ y = W.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}'); M$**:**

Assume $\Delta, x{:}T, \Delta'; \Sigma \vdash_{\bar{a}} \mathsf{let}\ y = W.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}'); M : \mathscr{T}\ \mathsf{Impure}$,

and, $\Delta \vdash V : T$.

By the type rule, $\Delta, x{:}T, \Delta'; \Sigma \vdash \diamond$,

and, $\Delta, x{:}T, \Delta' \vdash W : C$,

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

and, $\Delta, x{:}T, \Delta' \vdash \vec{\phi} : \vec{Q}$,

and, $\Delta, x{:}T, \Delta' \vdash \vec{W}' : \vec{T}'$,

and, $\vdash \vec{T}' <: \vec{T}[\vec{\beta} := \vec{\phi}]$,

and, $\Delta, x{:}T, \Delta', x{:}S[\vec{\beta} := \vec{\phi}], b{:}\mathsf{Prin}, b{=}W.\mathsf{loc}, \mathit{Prov}(x, b, a); \Sigma \vdash_{\bar{a}} M : \mathscr{T}\ \rho$.

By IH, $\Delta, \Delta'[x := V] \vdash W[x := V] : C[x := V]$,

and, $\Delta, \Delta'[x := V] \vdash \vec{\phi}[x := V] : \vec{Q}[x := V]$,

and, $\Delta, \Delta'[x := V] \vdash \vec{W}'[x := V] : \vec{T}'[x := V]$.

By Lemma 9, $\mathit{body}(C[x := V].\ell) = (\texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S(\vec{T}))[x := V]$.

By the type rule,

$\Delta, \Delta'[x := V]; \Sigma \vdash_{\bar{a}} \mathsf{let}\ y{=}W[x := V].\ell\texttt{<}\vec{\phi}[x := V]\texttt{>}(\vec{W}'[x := V]); M[x := V] : \mathscr{T}[x := V]\ \mathsf{Impure}$.

Case $W.\boldsymbol{f}_i$**:**

Assume $\Delta, x{:}T, \Delta'; \Sigma \vdash_{\bar{a}} W.f_i : \mathscr{T}\ \mathsf{Impure}$ *and* $\Delta \vdash V : T$.

By the type rule, $\Delta, x{:}T, \Delta' \vdash W : C$,

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

where $\mathscr{T} = T_i$.

By the Lemma 15, $\Delta, \Delta'[x := V] \vdash W[x := V] : C[x := V]$.

From the def., $\mathit{fields}(C[x := V]) = \vec{\mu}\ \vec{T}[x := V]\ \vec{f}$.

By the type rule, $\Delta, \Delta'[x := V] \vdash W[x := V].f_i : T_i[x := V]\ \mathsf{Impure}$.

Case $W.\boldsymbol{f}_i{:=}W'$**:**

Assume $\Delta, x{:}T, \Delta'; \Sigma \vdash_{\bar{a}} W.f_i{:=}W : \mathscr{T}\ \mathsf{Impure}$, *and* $\Delta \vdash V : T$.

By the type rule, $\mathscr{T} = \mathsf{Unit}$, *and,* $\Delta, x{:}T, \Delta' \vdash W : C$,

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

and, $\mu_i = \mathsf{mutable}$,

and, $\Delta, x{:}T, \Delta' \vdash W' : T_i'$,

and, $\vdash T_i' <: T_i$.

By Lemma 15, $\Delta, \Delta'[x := V] \vdash W[x := V] : C[x := V]$.

From the def., $\mathit{fields}(C[x := V]) = \vec{\mu}\ \vec{T}[x := V]\ \vec{f}$.

By Lemma 15, $\Delta, \Delta'[x := V] \vdash W'[x := V] : T_i'[x := V]$.

By Lemma 8, $\vdash T_i'[x := V] <: T_i[x := V]$.

By the type rule, $\Delta, \Delta'[x := V]; \Sigma \vdash_{\bar{a}} W[x := V].f_i{:=}W'[x := V] : \mathsf{Unit\ Impure}$. □

Theorem 5 (Type Preservation by Evaluation). *Suppose* $\Delta; \Sigma \vdash_{\bar{b}} M : \mathscr{T}\ \mathsf{Impure}$ *and* $M \rightarrow M'$. *Then* $\Delta; \Sigma \vdash_{\bar{b}} M' : \mathscr{T}\ \mathsf{Impure}$.

Proof. *By induction on the derivation of* $M \rightarrow M'$. *The pure term cases follow directly from lemma 17 and the cases for impure terms that have matching type rules use the same proofs as in lemma 17 but with the trivial addition of a store. The cases for the remaining impure terms are shown here.*

Case $a\texttt{[}\mathsf{new}\ c(\vec{V})\texttt{]} \rightarrow (\nu p)(a\texttt{[}p{:}c\texttt{\{}\vec{\boldsymbol{f}}{=}\vec{V}\texttt{\}]} \parallel a\texttt{[}p\texttt{]})$**:**

By hypothesis, $\mathit{fields}(c) = \vec{f}$ *and* $|\vec{f}| = |\vec{V}|$.
Assume $\Delta;\Sigma \vdash_{\overline{b}} a\,[\mathsf{new}\ c\,(\vec{V})\,] : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\mathscr{T}$ *has the form* $c\texttt{<}\vec{\phi}\texttt{>}$ *for some* $\vec{\phi}$,
  and, $\Delta;\Sigma \vdash_{\overline{a}} \mathsf{new}\ c\,(\vec{V}\,) : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta;\Sigma \vdash \diamond$,
  and, $\Delta \vdash c\texttt{<}\vec{\phi}\texttt{>}$,
  and, $\mathit{fields}(c\texttt{<}\vec{\phi}\texttt{>}) = \vec{\mu}\ \vec{T}\ \vec{f}$,
  and, $\Delta;\Sigma \vdash_{\overline{a}} \vec{V} : \vec{T}'\ \mathsf{Impure}$,
  and, $\vdash \vec{T}' <: \vec{T}$,
  and, $\mathit{effect}(c\texttt{<}\vec{\phi}\texttt{>}) = \theta$,
  and, $\Sigma \parallel a\,[\mathsf{this}\!:\!c\{\vec{f}\!=\!\vec{V}\}]\ \parallel \theta \Downarrow \psi$,
  and, $\mathit{clauses}(\Delta) \vDash a\ \mathit{says}\ \psi$.
By the type rule, $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>};\Sigma \vdash_{\overline{a}} a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] : \mathsf{Proc\ Impure}$.
By the type rule, $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>} \vdash p : c\texttt{<}\vec{\phi}\texttt{>}$.
By the type rule, $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>};\Sigma \vdash_{\overline{a}} p : c\texttt{<}\vec{\phi}\texttt{>}\ \mathsf{Impure}$.
By the type rule, $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>};\Sigma \vdash_{\overline{b}} a\,[p] : c\texttt{<}\vec{\phi}\texttt{>}\ \mathsf{Impure}$.
By def. of env, $\mathit{env}(a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]\,) = \cdot$,
  and, $\mathit{env}(a\,[p]\,) = \cdot$.
By def. of heap, $\mathit{heap}(a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]\,) = a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]$,
  and, $\mathit{heap}(a\,[p]\,) = \cdot$.
By weakening,
  $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>};\Sigma, a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] \vdash_{\overline{b}} a\,[p] : c\texttt{<}\vec{\phi}\texttt{>}\ \mathsf{Impure}$.
By type rule, $\Delta, p\!:\!c\texttt{<}\vec{\phi}\texttt{>};\Sigma \vdash_{\overline{b}} a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]\ \parallel a\,[p] : \mathscr{T}\ \mathsf{Impure}$.
By tp. rl., $\Delta;\Sigma \vdash_{\overline{b}} (\nu p\!:\!\mathscr{T}\,)(a\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]\ \parallel a\,[p]\,) : \mathscr{T}\ \mathsf{Impure}$.

Case $\begin{pmatrix} a_1\,[p\!:\!c\{\vec{\boldsymbol{f}}\!=\!\vec{V}\}]\ \parallel \\ a_2\,[\mathsf{let}\ y\!=\!p\,.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}\,)\,;L] \end{pmatrix} \to \begin{pmatrix} a_1\,[p\!:\!c\{\vec{\boldsymbol{f}}\!=\!\vec{V}\}]\ \parallel \\ a_2\,[\mathsf{let}\ y\!=\!a_1\,[M']\,;L'] \end{pmatrix}$**:**

By hypothesis, $\mathit{body}(c\,.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,(\vec{x}:\vec{T}\,)\{M\}$ *where* $|\vec{x}| = |\vec{V}|$,
  and, $M' = \mathit{Prov}(\vec{W}, a_2, a_1) \parallel M[\mathsf{caller} := a_2][\mathsf{this} := p][\vec{\beta} := \vec{\phi}][\vec{x} := \vec{W}]$,
  and, $L' = \mathit{Prov}(y, a_1, a_2) \parallel L$.
By Lemma 10, $\mathsf{caller}\!:\!\mathsf{Prin}, \mathsf{this}\!:\!c\texttt{<}\vec{\psi}\texttt{>}, \vec{\beta}:\vec{Q}, \vec{x}:\vec{T};\Sigma \vdash M : S\ \mathsf{Impure}$,
  where $\vec{\beta}$ *may be free in* S *and* $\vec{\beta}, \vec{x}, \mathsf{caller}$ *and* this *may be free in* M.
Assume $\Delta;\Sigma \vdash_{\overline{b}} a_1\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}]\ \parallel a_2\,[\mathsf{let}\ y\!=\!p\,.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}\,)\,;L] : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta;\Sigma \vdash_{\overline{b}} a_1\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] : \mathscr{T}'\ \mathsf{Impure}$,
  and, $\Delta;\Sigma, a_1\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] \vdash_{\overline{b}} a_2\,[\mathsf{let}\ y\!=\!p\,.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}\,)\,;L] : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta \vdash a_2 : \mathsf{Prin}$,
  and, $\Delta;\Sigma, a_1\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] \vdash_{\overline{a_2}} \mathsf{let}\ y\!=\!p\,.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W}\,)\,;L : \mathscr{T}\ \mathsf{Impure}$,
  and, $\mathscr{T} = S[\vec{\beta} := \vec{Q}]$.
By the type rule, $\Delta;\Sigma \vdash \diamond$,
  and, $\Delta \vdash p : c\texttt{<}\vec{\psi}\texttt{>}$,
  and, $\mathit{body}(c\,.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,(\vec{x}:\vec{T}\,)\{M\}$,
  and, $\Delta \vdash \vec{\phi} : \vec{Q}$,
  and, $\Delta \vdash \vec{W} : \vec{T}'$,
  and, $\vdash \vec{T}' <: \vec{T}[\vec{\beta} := \vec{\phi}]$,
  and, $\Delta, y\!:\!\_, b\!:\!\mathsf{Prin}, b\!=\!p\,.\mathsf{loc}, \mathit{Prov}(y, b, a_2);\Sigma, a_1\,[p\!:\!c\{\vec{f}\!=\!\vec{V}\}] \vdash_{\overline{a_2}} L : \mathscr{T}\ \mathsf{Impure}$,

where $b \notin fn(L, \mathscr{T})$.
By the type rule, $\Delta \vdash Prov(\vec{W}, a_2, a_1) : \_$.
By the type rule, $\Delta, y : \_, a_1 = p.\mathsf{loc} \vdash Prov(y, a_1, a_2) : \_$.
By substitution,
$\Delta, y : \_, a_1 = p.\mathsf{loc}, Prov(y, a_1, a_2); \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} L : \mathscr{T}$ Impure.
By the type rule, $\Delta, y : \_, a_1 = p.\mathsf{loc}; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} L' : \mathscr{T}$ Impure.
By Lemma 2, $\Delta, a_1 = p.\mathsf{loc}, y : \_; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} L' : \mathscr{T}$ Impure.
By Lemma 13, $\Delta, a_1 = a_1, y : \_; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} L' : \mathscr{T}$ Impure.
By Lemma 12, $\Delta, y : \_; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} L' : \mathscr{T}$ Impure.
By substitution,
$\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} M[\mathsf{caller} := a_2][\mathsf{this} := p][\vec{\beta} := \vec{\phi}][\vec{x} := \vec{W}] : \_$ Impure.
By the type rule, $\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} M' : \_$ Impure.
By the type rule,
$\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} \mathsf{let}\, y = a_1 [M']; L' : \mathscr{T}$ Impure.
By the type rule,
$\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{b}} a_2 [\mathsf{let}\, y = a_1 [M']; L'] : \mathscr{T}$ Impure.
By the type rule,
$\Delta; \Sigma \vdash_{\overline{b}} a_1 [p : c\{\vec{f} = \vec{V}\}] \Vdash a_2 [\mathsf{let}\, y = a_1 [M']; L'] : \mathscr{T}$ Impure.

Case $\begin{pmatrix} a_1 [p : c\{\vec{\boldsymbol{f}} = \vec{V}\}] \Vdash \\ a_2 [p.\boldsymbol{f}_i := W] \end{pmatrix} \to \begin{pmatrix} a_1 [p : c\{\vec{\boldsymbol{f}} = \vec{V}[V_i := W]\}] \Vdash \\ a_2 [\mathsf{unit}] \end{pmatrix}$**:**

Assume $\Delta; \Sigma \vdash_{\overline{b}} a_1 [p : c\{\vec{f} = \vec{V}\}] \Vdash a_2 [p.f_i := W] : \mathscr{T}$ Impure.
By def. of env, $env(a_1 [p : c\{\vec{f} = \vec{V}\}]) = \cdot$,
and, $env(a_2 [p.f_i := W]) = env(p.f_i := W) = \cdot$.
By def. of heap,
$heap(a_1 [p : c\{\vec{f} = \vec{V}\}]) = a_1 [p : c\{\vec{f} = \vec{V}\}]$,
and, $heap(a_2 [p.f_i := W]) = heap(p.f_i := W) = \cdot$.
By the type rule, $\mathscr{T}$ *is of the form* T,
and, $\Delta; \Sigma \vdash_{\overline{b}} a_1 [p : c\{\vec{f} = \vec{V}\}] : \mathscr{T}'$ Impure,
and, $\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{b}} a_2 [p.f_i := W] : \mathscr{T}$ Impure,
and, $fn(a_1 [p : c\{\vec{f} = \vec{V}\}] \Vdash a_2 [p.f_i := W]) \subseteq dom(\Delta)$.
By the type rule, $\Delta \vdash p : c\langle\vec{\phi}\rangle$,
and, $fields(c) = \vec{\mu}\ \vec{T}\ \vec{f}$,
and, $\Delta \vdash \vec{V} : \vec{T}$.
By the type rule, $\Delta \ni p : c\langle\vec{\phi}\rangle$.
By the type rule, $\Delta; \Sigma, a_1 [p : c\{\vec{f} = \vec{V}\}] \vdash_{\overline{a_2}} p.f := W : \mathscr{T}$ Impure.
By the type rule, $\mathscr{T} = \mathsf{Unit}$,
and, $\Delta \vdash p : C$,
and, $fields(C) = \vec{\mu}'\ \vec{T}'\ \vec{f}'$,
and, $\mu'_i = \mathsf{mutable}$,
and, $\Delta \vdash W : T''_i$,
and, $\vdash T''_i <: T'_i$.
It is easy to see that $C = c\langle\vec{\phi}\rangle$,
therefore, $\mu_i = \mathsf{mutable}$,
and, $\vdash T''_i <: T_i$.

By the type rule, $\Delta;\Sigma \vDash_{\vec{b}} a_1\,[p\!:\!c\{\vec{f}=\vec{V}[V_i := W]\}]\ :\ \mathscr{T}'\ \mathsf{Impure}$.
By the type rule, $\Delta \vdash \mathsf{unit} : \mathscr{T}$.
By the type rule, $\Delta;\Sigma, a_1\,[p\!:\!c\{\vec{f}=\vec{V}[V_i := W]\}] \vDash_{a_2} \mathsf{unit} : \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta;\Sigma, a_1\,[p\!:\!c\{\vec{f}=\vec{V}[V_i := W]\}] \vDash_{\vec{b}} a_2\,[\mathsf{unit}]\ :\ \mathscr{T}\ \mathsf{Impure}$.
By the type rule, $\Delta;\Sigma \vDash_{\vec{b}} a_1\,[p\!:\!c\{\vec{f}=\vec{V}[V_i := W]\}] \parallel a_2\,[\mathsf{unit}]\ :\ \mathscr{T}\ \mathsf{Impure}$. □

Theorem 6 (Progress). *Suppose* $\vdash M : T\ \mathsf{Impure}$. *Then either* $M \rightarrow N$ *or* $M \equiv (\nu\vec{p}\!:\!\vec{C})\,(M_1 \parallel \ldots \parallel M_n \parallel V)$ *where for all i,* M_i *is either a value or an inert process.*

Proof. By Proposition 1 all terms are equivalent to a term in normal for, so we assume w.l.o.g that M is in normal form. Suppose $\vdash (\nu\vec{p}\!:\!\vec{C})\,(N \parallel M') : T\ \mathsf{Impure}$ *where* $N = (W_1 \parallel \cdots \parallel W_\ell \parallel \mathbb{N}_1 \parallel \cdots \parallel \mathbb{N}_m \parallel b_1\,[\mathbb{L}_1] \parallel \cdots \parallel b_n\,[\mathbb{L}_n])$ *and* M' *is of the form V or* $\mathbb{N}$ *or* $a\,[\mathbb{L}]$. *We use* N_i *to denote the ith component of N. By the type rule,* $\vec{p}\!:\!\vec{C} \vdash N \parallel M' : T\ \mathsf{Impure}$. *By the type rule,* $\vec{p}\!:\!\vec{C}; heap(N) \vdash M' : T\ \mathsf{Impure}$. *We first show that either M can evaluate, or* M' *is a value by induction on the structure of* M':

Case V**:**
A value.

Case $\mathbb{N}$**:**

Subcase $\mathsf{new}\ c\langle\vec{\phi}\rangle(\vec{V})$**:**
Term can evaluate.

Subcases $\mathsf{let}\ y = V.\ell\langle\vec{\phi}\rangle(\vec{W});\ M\ \mid\ V.\boldsymbol{f}\ \mid\ V.\mathsf{loc}\ \mid\ V.\boldsymbol{f} := W$**:**
By the type rules for each of these terms, $\vec{p}\!:\!\vec{C}; heap(N) \vdash \diamond$,
and, $\vec{p}\!:\!\vec{C} \vdash V : C$,
By the type rule, $\vec{p}\!:\!\vec{C} \ni V : C$.
By the rule for w.f.e., $(\exists H) heap(N) \ni H$ *and* $H = V : c\{\vec{f} = \vec{W}\}$.
From the def of heap, this can only be if $(\exists i) N_i = V : c\{\vec{f} = \vec{W}\}$.
But then M would be able to evaluate.

Subcase $\mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N$**:**
Term can evaluate.

Subcase $\mathsf{let}\ x = \mathbb{N};\ L$**:**
By the structural rule and Theorem 3,
$\vec{p}\!:\!\vec{C} \vdash \mathsf{let}\ x = (N \parallel \mathbb{N});\ L : T\ \mathsf{Impure}$.
By the structural rule,
$\vdash \mathsf{let}\ x = (\nu\vec{p}\!:\!\vec{C})\,(N \parallel \mathbb{N});\ L : T\ \mathsf{Impure}$.
By the type rule,
$\vdash (\nu\vec{p}\!:\!\vec{C})\,(N \parallel \mathbb{N}) : T\ \mathsf{Impure}$.
By IH, either $\mathbb{N}$ *is a value, in which case the term can evaluate directly,*
or $(N \parallel \mathbb{N})$ *can evaluate, in which case it can evaluate by the context rule.*

Subcase $\mathsf{let}\ x = V;\ M$**:**
Term can evaluate.

Subcase $p : c\{\vec{\boldsymbol{f}} = \vec{V}\}$**:**
Not applicable; cannot type as T.

Case $a\,[\mathbb{L}]$**:**
All subcases are the same as for $\mathbb{N}$ *modulo an application of a structural rule or a context rule.*

Now consider each N_i. The structural rules can be used to rewrite M as $(\nu\vec{p}:\vec{C})(M^i \Vdash N_i \Vdash M')$ where $M^i = (N_1 \Vdash \ldots \Vdash N_{i-1} \Vdash N_{i+1} \Vdash \ldots \Vdash N_{n-1})$. By the type rule, $\vec{p}:\vec{C} \vdash M^i \Vdash N_i \Vdash M' : T$ Impure. By the type rule, noting that the structure of M' implies that $env(M') = heap(M') = \emptyset$, $\vec{p}:\vec{C}; heap(M^i) \vdash N_i : \mathscr{T}$ Impure. We now show that either each N_i is a value or an inert process, or M can evaluate by induction on the structure of N_i. All cases are essentially identical to the previous inductive proof, except that $p:c\{\vec{f}=\vec{V}\}$ is allowed because it is an inert processes. □

Definition 5 (Safety). *Define let-contexts $\mathbb{E}$ as*

$$\mathbb{E} ::= [] \mid \mathsf{let}\ x = \mathbb{E}; M$$

A term M is safe *if whenever*

$$M \to^{*}\equiv (\nu\vec{p}:\vec{C})\, a\,[\mathbb{E}[\mathsf{new}\ c\texttt{<}\vec{\phi}\texttt{>}(\vec{V})]] \Vdash M'$$

or

$$M \to^{*}\equiv (\nu\vec{p}:\vec{C})\, M' \Vdash a\,[\mathbb{E}[\mathsf{new}\ c\texttt{<}\vec{\phi}\texttt{>}(\vec{V})]],$$

and $effect(c\texttt{<}\vec{\phi}\texttt{>}) = \theta$ and $heap(M') = \Sigma$ and $(\Sigma \Vdash a\,[p:c\texttt{<}\vec{\phi}\texttt{>}\{\vec{V}\}]) \Vdash \theta[\mathsf{this} := p] \Downarrow \psi$ (where $p \notin fn(\theta)$) then either $clauses(\vec{p}:\vec{C}, env(M')) \vDash \psi$ or $a = \mathbf{1}$.

Corollary 2 (Safety). *Suppose that $\vec{p}:\vec{C};\Sigma \vdash_{a} M : T$ Impure. Then $(\nu\vec{p}:\vec{C})\,\mathbf{1}\,[N] \Vdash \Sigma \Vdash a\,[M]$ is safe for any $N, \mathscr{T}$ such that $\vec{p}:\vec{C};\Sigma \vdash_{\mathbf{1}} N : \mathscr{T}$ Impure.*

Proof. *A corollary of Theorem 5.*