Obsidian: Typestate and Assets for Safer Blockchain Programming

MICHAEL COBLENZ, Carnegie Mellon University, USA
REED OEI*, University of Illinois at Urbana-Champaign, USA
TYLER ETZEL*, Facebook, USA
PAULETTE KORONKEVICH*, University of British Columbia, Canada
MILES BAKER*, Amazon, USA
YANNICK BLOEM*, Apple, Inc, USA
BRAD A. MYERS, JOSHUA SUNSHINE, and JONATHAN ALDRICH, Carnegie Mellon University, USA

Blockchain platforms are coming into use for processing critical transactions among participants who have not established mutual trust. Many blockchains are programmable, supporting *smart contracts*, which maintain persistent state and support transactions that transform the state. Unfortunately, bugs in many smart contracts have been exploited by hackers. Obsidian is a novel programming language with a type system that enables static detection of bugs that are common in smart contracts today. Obsidian is based on a core calculus, Silica, for which we proved type soundness. Obsidian uses *typestate* to detect improper state manipulation and uses *linear types* to detect abuse of assets. We integrated a permissions system that encodes a notion of *ownership* to allow for safe, flexible aliasing. We describe two case studies that evaluate Obsidian's applicability to the domains of parametric insurance and supply chain management, finding that Obsidian's type system facilitates reasoning about high-level states and ownership of resources. We compared our Obsidian implementation to a Solidity implementation, observing that the Solidity implementation requires much boilerplate checking and tracking of state, whereas Obsidian does this work statically.

*Work performed while at Carnegie Mellon University.

This material is based upon work supported by the National Science Foundation under Grants CNS-1423054 and CCF-1814826, by the U.S. Department of Defense, and by Ripple. In addition, the first author is supported by an IBM Ph.D. Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Authors' addresses: M. Coblenz, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA 15213; email: mcoblenz@cs.cmu.edu; R. Oei, Computer Science Department, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue MC 258, Urbana, IL 61801; email: reedoei2@illinois.edu; T. Etzel, Facebook, 1 Hacker Way, Menlo Park, CA 94025; email: tyleretzel1@gmail.com; P. Koronkevich, Computer Science Department, University of British Columbia, 2329 West Mall, Vancouver, BC, V6T 1Z4, Canada; email: pletrec@cs.ubc@ca; M. Baker, Amazon, 410 Terry Ave. North, Seattle, WA 98109; email: milesabaker@gmail.com; Y. Bloem, Apple, Inc, One Apple Park Way, Cupertino, CA 95014; email: yannickbloem@gmail.com; B. A. Myers, Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA 15213; email: bam@cs.cmu.edu; J. Sunshine and J. Aldrich, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA 15213; emails: {joshua.sunshine, jonathan.aldrich}@cs.cmu.edu.

0164-0925/2020/11-ART14
https://doi.org/10.1145/3417516

CCS Concepts: • **Software and its engineering → Language features**; **Domain specific languages**; • **Security and privacy** → *Software and application security*;

Additional Key Words and Phrases: Typestate, linearity, type systems, blockchain, smart contracts, permissions, alias control, ownership

ACM Reference format:
Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2020. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 14 (November 2020), 82 pages.
https://doi.org/10.1145/3417516

1 INTRODUCTION

Blockchains have been proposed to address security and robustness objectives in contexts that lack shared trust. By recording all transactions in a tamper-resistant *ledger*, blockchains attempt to facilitate secure, trusted computation in a network of untrusted peers. Blockchain programs, sometimes called *smart contracts* [Szabo 1997], can be deployed; once deployed, they can maintain state in the ledger. For example, a program might represent a bank account and store a quantity of virtual currency. Clients could conduct transactions with bank accounts by invoking the appropriate interfaces. Each transaction is appended permanently to the ledger. In this article, we refer to a deployment of a smart contract as an *object* or *contract instance*.

Proponents have suggested that blockchains be used for a plethora of applications, such as finance, health care [Harvard Business Review 2017], supply chain management [IBM 2019], and others [Elsden et al. 2018]. For example, an electronics manufacturer might accept shipments of components from a variety of manufacturers; if any of those components have been replaced with fraudulent components somewhere in the chain of custody, then the manufactured systems might include defects, including security vulnerabilities [Dieterich et al. 2017]. A blockchain could provide a tamper-resistant mechanism for recording signed transactions showing every entity that was ever responsible for each component.

Unfortunately, some prominent blockchain applications have included security vulnerabilities. For example, the DAO and Parity bugs were exploited to steal over $80 million worth of virtual currency [Graham 2017; Sirer 2016]. In addition to the potentially severe consequences of bugs, platforms require that contracts be immutable, so bugs cannot be fixed easily. If organizations are to adopt blockchain environments for business-critical applications, then there needs to be a more reliable way of writing smart contracts.

Many techniques promote program correctness, but our focus is on programming language design so that we can prevent bugs as early as possible—potentially by aiding the programmer's reasoning processes before code is even written. Because of our interest in developing a language that would be effective for programmers, we designed a surface language, *Obsidian*, in addition to a core calculus, *Silica*. Obsidian stands for *Overhauling Blockchains with States to Improve Development of Interactive Application Notation*. Our design is based on formative studies with programmers, and although those studies are not the focus of this article, our goal of *usability* drove us to focus on features that provide powerful safety guarantees while maintaining as much simplicity as possible. In this article, we focus on the design of the language itself and make only brief mention of our observations in our user studies. For more detail regarding the user studies, readers may refer to Coblenz et al. [2020a, 2019a].

Obsidian is a programming language for smart contracts that provides strong compile-time features to prevent bugs. Obsidian is based on a novel type system that uses *typestate* [Strom and

Yemini 1986] to statically ensure that objects are manipulated correctly according to their current states and uses *linear types* [Wadler 1990] to enable safe manipulation of assets, which must not be accidentally lost. We prove key soundness theorems so that Silica can serve as a trustworthy foundation for Obsidian and potentially other typestate-oriented languages.

We make the following contributions:

(1) We show how typestate and linear types can be combined in a user-facing programming language, using a rich but simple permission system that captures the required restrictions on aliases using a notion of ownership.
(2) We show an integrated architecture for supporting both smart contracts and client programs. By enabling both on-blockchain and off-blockchain programs to be created with the same language, we ensure that the safety properties of the language are available for data structures that must be transferred off-blockchain as well as for those stored in the blockchain.
(3) We describe Silica, the core calculus that underlies Obsidian. We prove type soundness and asset retention for Silica. Asset retention is the property that owning references to assets (objects that the programmer has designated have value) cannot be lost accidentally. Silica is the first typestate calculus (of which we are aware) that supports assets.
(4) As case studies, we show how Obsidian can be used to implement a parametric insurance application and a supply chain. Through a comparison to Solidity, we show how leveraging typestate can move checks from execution time to compile time. Our case studies were implemented by programmers who were not the designers of the language, showing that the language is usable by people other than only the designers.

Obsidian is available on GitHub (http://www.github.com/mcoblenz/Obsidian); an archive is available as well [Coblenz et al. 2020b].

After summarizing related work, we introduce the Obsidian language with an example (Section 3). Section 4 focuses on the design of particular aspects of the language and describes how qualitative studies influenced our design. We describe how the language design fits into the Fabric blockchain infrastructure in Section 5. Section 6 describes Silica, the core calculus underlying Obsidian, and its proof of soundness (although the proof itself is in Appendix C). After discussing how the full Obsidian language extends Silica (Section 7), we discuss two case studies in Section 8, showing how we have collaborated with external stakeholders to demonstrate the expressiveness and utility of Obsidian. Future work is discussed in Section 9. We conclude in Section 10.

2 RELATED WORK

2.1 Smart Contract Languages

Solidity [Ethereum Foundation 2020a], which targets the Ethereum platform [Ethereum Foundation 2020b], is the predominant domain-specific smart contract language (some blockchain platforms support general-purpose languages; for example, Hyperledger Fabric supports Go, Java, and JavaScript). Researchers have previously investigated common causes of bugs in smart contracts [Atzei et al. 2017; Delmolino et al. 2016; Luu et al. 2016], created static analyses for existing languages [Feist et al. 2019; Grech et al. 2020; Kalra et al. 2018], and worked on applying formal verification techniques [Alt and Reitwiessner 2018; Bhargavan et al. 2016; Zakrzewski 2018]. Our work focuses on preventing bugs in the first place by designing a language in which many commonplace bugs can be prevented as a result of properties of the type system. This enables programmers to reason more effectively about relevant safety properties and enables the compiler to detect many relevant bugs.

There is a large collection of proposals for new smart contract languages, cataloged by Harz and Knottenbelt [2018]. One of the languages most closely related to Obsidian is Flint [Schrans et al. 2019]. Flint supports a notion of typestate but lacks a permission system that, in Obsidian, enables flexible, static reasoning about aliases. Flint supports a trait called `Asset`, which enhances safety for resources to protect them from being duplicated or destroyed accidentally. However, Flint models assets as traits rather than as linear types due to the aliasing issues that this would introduce [Schrans and Eisenbach 2019]. This leads to significant limitations on assets in Flint. For example, in Flint, assets cannot be returned from functions. Obsidian addresses these issues with a permission system and thus permits any non-primitive type to be an asset and treated as a first-class value.

Scilla [Sergey et al. 2019] is an intermediate language for smart contracts. Like Obsidian, Scilla is oriented around state machines. However, in contrast with Obsidian, Scilla is a functional language and is intended as a target for compilers, not as a high-level language in which programmers can be effective. Scilla offers restricted computation: no loops, no effectful recursion, and no calls to mutating transactions. Although this may suffice for many smart contracts, Obsidian provides a richer, Turing-complete environment. Likewise, IELE [Kasampalis et al. 2019] is an intermediate language with a compiler that translates from Solidity. IELE is intended to facilitate automatic formal verification of properties that are specified in the K framework [Roşu and Şerbănuţă 2010].

There are also proposals for blockchain languages that are more domain specific. For example, Hull et al. propose formalizing a notion of business artifacts for blockchains [Hull et al. 2016]. DAML [Digital Asset, Inc. 2019] is more schema-oriented, requiring users to write schemata for their data models. In DAML, which was inspired by financial agreements, contracts specify who can conduct and observe various operations and data. Likewise, Astigarraga et al. [2018] proposes a rules-based language to enable business users with less programming background to author smart contracts.

Xu et al. [2017] gives a taxonomy of blockchain systems. There, the focus is on blockchain platforms, i.e., systems that maintain blockchains and process transactions. The architecture of a particular blockchain platform can have some implications on application architecture and design.

2.2 Aliasing, Permissions, and Linearity

The problem of aliasing in object-oriented languages has led to significant research on ways to constrain and reason about aliases [Clarke et al. 2013]. Unfortunately, these approaches can be very complex. For example, fractional permissions [Boyland 2003] provide an algebra of permissions to memory cells. These permissions can be split among multiple references so that if the references are combined, one can recover the original (whole) permission. However, aside from the simple approach of reference counting, general fractional permissions have not been adopted in practical languages, perhaps because using them requires understanding a complex algebraic system.

A significant line of research has focused on *ownership types* [Clarke et al. 1998], which refers to a different notion of ownership than we use in Obsidian. Ownership types aim to enforce *encapsulation* by ensuring that the implementation of an object cannot leak outside its owner. In Obsidian, we are less concerned with encapsulation and more focused on sound typestate semantics. This allows us to avoid the strict nature of these encapsulation-based approaches while accepting their premise: Typically, good architecture results in an aliasing structure in which one "owner" of a particular object controls the object's lifetime and, likely, many of the changes to the object.

Gordon et al. [2012] describes a type system that uses permissions to enable safe concurrency. This work focuses on concurrency but does not help with reasoning about object protocols (as typestate does). Although the significant restrictions that are there to handle concurrency are warranted in those contexts, because blockchain systems are now always sequential, this complexity is

not needed for Obsidian. For example, *isolated* references in Gordon et al. [2012] do not allow read-only (*readable*) aliases to mutable objects reachable from the isolated references; owned references in Obsidian have no such restriction, because Obsidian does not need to support concurrency.

Linear types, which facilitate reasoning about *resources*, have been studied in depth since Wadler's 1990 paper [Wadler 1990] but have not been adopted in many programming languages. Rust [Mozilla Research 2015] is one exception, using a form of linearity to restrict aliases to mutable objects. This limited use of linearity did not require the language to support as rich a permission system as Obsidian does; for example, Rust types cannot directly express states of referenced objects. Alms [Tov and Pucella 2011] is an ML-like language that supports linear types; unlike Obsidian, it is not object oriented. Session types [Caires and Pfenning 2010] are another way of approaching linear types in programming languages, as in Concurrent C0 [Willsey et al. 2017]. However, session types are more directly suited for communicating, concurrent processes, which is very different from a sequential, stateful setting as is the case with blockchains.

2.3 Typestate

Fickle [Drossopoulou et al. 2002] was one approach to allow objects to change class at execution time, but Fickle did not allow references to include any type specifications pertaining to the states of the referenced objects. DeLine investigated using typestate in the context of object-oriented systems [DeLine and Fähndrich 2004], finding that subclassing causes complicated issues of partial state changes; we avoid that problem by not supporting subclassing. Plaid [Sunshine et al. 2011] and Plural [Bierhoff and Aldrich 2008] are the most closely related systems in terms of their type systems' features. Both languages were complex, and the authors noted the complexity in certain cases, e.g., fractional permissions make the language harder to use but were rarely used and even then primarily for concurrency [Bierhoff et al. 2011]. Sunshine et al. [2014] showed typestate to be helpful in documentation when users need to understand object protocols; we used that conclusion as motivation for our language design.

Silica (Section 6), the core of Obsidian, is based on Featherweight Typestate (FT) [Garcia et al. 2014]. However, since Silica is designed as the core of a user-facing programming language, there are significant differences, because we wanted our core language to allow us to formalize particular operations that we included for usability reasons in the surface language. For example, Silica replaces FT's atomic field swap with field assignment. This allows fields that temporarily have modes that differ from their declarations, facilitating a style of programming that our participants preferred in our formative user studies. This approach is related to the approach taken in Naden et al. [2012], where fields can be unpacked, but in Silica, unpacking is only possible via the this reference to maintain encapsulation. Key differences between Silica and FT are shown in Table 1.

Silica avoids class-level inheritance to simplify reasoning about programs. To formalize expression of (a) fields that are common to more than one state and (b) a type system that is aware of all possible (nominal) states of a particular object, Silica defines a notion of *state* in addition to a notion of *contract*. FT only has a notion of *class* and expects the programmer to simulate states by specifying multiple classes that interoperate.

Support for dynamic state tests is an important feature to facilitate practical programming. Support for these dynamic state tests has been found to be critical for expressiveness in other object-oriented contexts as well [Bierhoff et al. 2009]. Unlike FT, Silica supports expressions that execute dynamic state tests so that programs can branch according to the result.

Silica fuses the notions of typestate and permission into one type construct, unlike FT, which has separate notions of permission and state guarantee. This approach allows the syntax of Silica to exactly express the set of possible reference types. Silica also distinguishes between asset contracts and non-asset contracts; owning references to asset contracts are treated *linearly* rather than in

Table 1. Key Differences between Featherweight Typestate and Silica

Featherweight Typestate	Silica
`pure` references cannot be used to mutate fields of referenced objects	Unowned references are only as restricted as necessary for soundness: they cannot be used to change nominal state but can be used to write fields
No dynamic state tests	Dynamic state tests
Types integrate state guarantees, but do not separate state from class	Separate contract and state constructs
Inheritance	No inheritance
Typestate is integrated with class	Typestate implies ownership
No linear assets	Linear assets with explicit disown

an affine way (i.e., linear references must be conserved, whereas affine references can be lost). FT has no way of treating references linearly.

Silica supports parametric polymorphism, a key feature requested by our industrial stakeholders. Although this makes the language more complex, we think this complexity is outweighed by the benefit of the feature, enabling (for example) reusable containers.

As a result, although some aspects of the system are more complex than FT, Silica is more expressive in the above respects. Silica serves as a sound foundation for Obsidian but could be used or adapted for other typestate-oriented languages.

Although the user-centered design aspects of Obsidian [Coblenz et al. 2019a] are not the focus of this article, others have had success applying user-centered methods to tools for developers. For example, Myers et al. [2016] argued that human-centered methods could be used in a variety of different tools for software engineers. Pane, Myers, and Miller used HCI techniques to design a programming language for children [Pane et al. 2002]. Stefik and Siebert used an empirical, quantitative approach regarding the design of syntax [Stefik and Siebert 2013].

3 INTRODUCTION TO THE OBSIDIAN LANGUAGE

Obsidian is based on several guidelines for the design of smart contract languages that we identified in Coblenz et al. [2019b]. Briefly, those guidelines are as follows:

- Strong static safety: Bugs are particularly serious when they occur in smart contracts. In general, it can be impossible to fix bugs in deployed smart contracts because of the immutable nature of blockchains. Obsidian emphasizes a novel, strong, static type system to detect important classes of bugs at compile time. Among common classes of bugs is *loss of assets*, such as virtual currency.
- User-centered design: A proposed language should be as usable as possible. We integrated feedback from users to maximize users' effectiveness with Obsidian.
- Blockchain-agnosticism: Blockchain platforms are still in their infancies and new ones enter and leave the marketplace regularly. Being a significant investment, a language design should target properties that are common to many blockchain platforms.

We were particularly interested in creating a language that we would eventually be able to evaluate with users, while at the same time significantly improving safety relative to existing language designs. In short, we aimed to create a language that we could show was more effective for pro-

grammers. To make this practical, we made some relatively standard surface-level design choices that would enable our users to learn the core language concepts more easily, while using a sophisticated type system to provide strong guarantees. Where possible, we chose approaches that would enable static enforcement of safety, but in a few cases we moved checks to execution time to enable a simple design for users or a more precise analysis (for example, in dynamic state checks, Section 4.5).

Typestate-oriented programming [Aldrich et al. 2009] has been proposed to allow specification of protocols in object-oriented settings. For example, a `File` can only be read when it is in the `Open` state, not when it is in the `Closed` state. By lifting these specifications into types, typestate-oriented programming languages allow static checking of adherence to protocols and improve the ability of programmers to reason effectively about how to use the interfaces correctly [Sunshine et al. 2014]. Featherweight Typestate [Garcia et al. 2014] is a core calculus for a class of typestate languages. However, we found in user studies that our early prototypes of Obsidian, which were based on a simplified version of this calculus, led to significant user confusion [Coblenz et al. 2019a]. To address these problems, we elicited blockchain language requirements from blockchain application implementations and proposals [Coblenz et al. 2019b]. These requirements motivated the design of a new formalism; we designed *Silica*, a new typestate calculus that, despite its simplicity, still allows users to express nearly all the properties that earlier typestate calculi enabled. Silica also supports key features that we observed users expected to have, such as dynamic state tests and field assignment.

We selected an object-oriented approach, because smart contracts inevitably implement state that is mutated over time, and object-oriented programming is well known to be a good match to this kind of situation. This approach is also a good starting point for our users, who likely have some object-oriented programming experience. However, to improve safety relative to traditional designs, Obsidian omits inheritance, which is error-prone due to the *fragile base class problem* [Mikhajlov and Sekerinski 1998], in which seemingly innocuous changes to a base class can break derived classes. We leveraged some features of the C-family syntax, such as blocks delimited with curly braces, dots for separating references from members, and so on, to improve learnability for some of our target users. Following blockchain convention, Obsidian uses the keyword `contract` rather than `class`. Because of the transactional semantics of invocations on blockchain platforms, Obsidian uses the term `transaction` rather than `method`. Transactions can require that their arguments, including the receiver, be in specific states in order for the transaction to be invoked.

Since smart contracts frequently manipulate assets, such as cryptocurrencies, we designed Obsidian to support linear types [Wadler 1990], which allow the compiler to ensure that assets are neither duplicated nor lost accidentally. These linear types integrate consistently with typestate, since typestate-bearing references are affine (i.e., cannot be duplicated but can be dropped as needed). A particular innovation in this approach is the fusion of linear references to assets with affine references to non-assets. Whether a reference is linear or affine depends on the declaration of the type to which the reference refers.

The example in Figure 1 shows some of the key features of Obsidian. `TinyVendingMachine` is a `main` contract, so it can be deployed independently to a blockchain. A `TinyVendingMachine` has a very small inventory: just one candy bar. It is either `Full`, with one candy bar in inventory, or `Empty`. Clients may invoke `buy` on a vending machine that is in `Full` state, passing a `Coin` as payment. When `buy` is invoked, the caller must initially *own* the `Coin`, but after `buy` returns, the caller no longer owns it. `buy` returns a `Candy` to the caller, which the caller then owns. After `buy` returns, the vending machine is in state `Empty`.

```
1  // This vending machine sells candy in exchange for coins.
2  main asset contract TinyVendingMachine {
3      // Fields defined at the top level are in scope in all states.
4      Coins @ Owned coinBin;
5
6      state Full {
7          // inventory is only in scope when the object is in Full state.
8          Candy @ Owned inventory;
9      }
10     state Empty; // No candy if the machine is empty.
11
12     TinyVendingMachine() {
13         coinBin = new Coins(); // Start with an empty coin bin.
14         ->Empty; // start in the Empty state
15     }
16
17     // this must be in the Empty state to call restock, and
18     //   this transitions to Full state
19     // c references a Candy that is initially Owned by the caller, and ends up
20     //   Unowned by the caller since it is owned by this
21     transaction restock(TinyVendingMachine @ Empty >> Full this,
22                         Candy @ Owned >> Unowned c) {
23         // transition to the Full state with c as the inventory
24         ->Full(inventory = c);
25     }
26
27     transaction buy(TinyVendingMachine @ Full >> Empty this,
28                     Coin @ Owned >> Unowned c) returns Candy @ Owned {
29         coinBin.deposit(c);
30         Candy result = inventory;
31         ->Empty;
32         return result;
33     }
34
35     transaction withdrawCoins() returns Coins @ Owned {
36         Coins result = coinBin;
37         coinBin = new Coins();
38         return result;
39     }
40 }
```

Fig. 1. A tiny vending machine implementation, showing key features of Obsidian.

Smart contracts commonly manipulate *assets*, such as virtual currencies. Some common smart contract bugs pertain to accidental loss of assets [Delmolino et al. 2016]. If a contract in Obsidian is declared with the `asset` keyword, then the type system requires that every instance of that contract have exactly one owner. This enables the type checker to report an error if an owned reference goes out of scope. For example, assuming that `Coin` was declared as an `asset`, if the author of the buy transaction had accidentally omitted the `deposit` call, then the type checker would have reported the loss of the asset in the buy transaction. Any contract that has an `Owned` reference to another asset must itself be an asset.

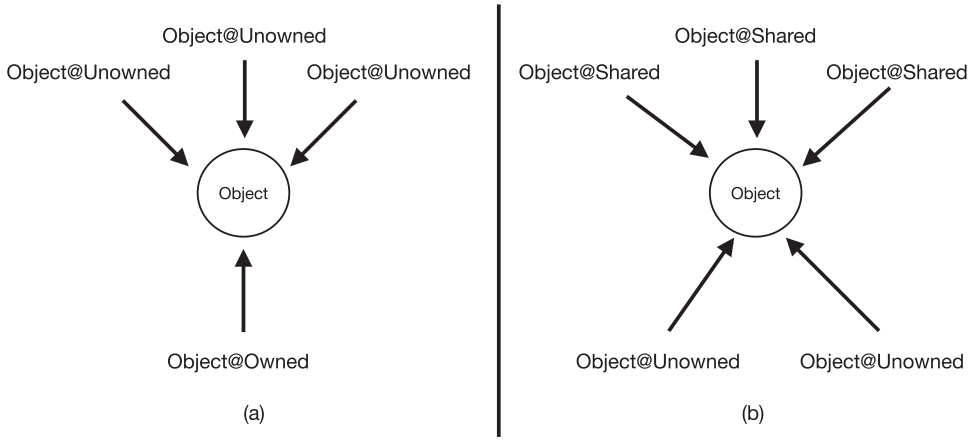

Fig. 2. Some common aliasing scenarios. (a) An object with one owner; (b) a shared object.

Table 2. A Summary of Modes in Obsidian

Mode	Meaning	Typestate mutation
Owned	This is the only reference to the object that is owned. There may be many Unowned aliases but no Shared aliases.	Permitted
Unowned	There may or may not be any owned aliases to this object, but there may be many other Unowned or Shared aliases.	Forbidden
Shared	This is one of potentially many Shared references to the object. There are no owned aliases.	Permitted
state name(s)	This is an owned reference and also conveys the fact that the referenced object is in one of the specified states. There may be Unowned aliases but no Shared or Owned aliases.	Permitted

References to objects have types according to both the `contract` of the referenced object and a *mode*, which denotes information about ownership. Modes are separated from contract names with an `@` symbol. Exactly one reference to each asset contract instance must be Owned; this reference must not go out of scope. For example, an owned reference to a `Coin` object can be written `Coin@Owned`. Ownership can be transferred between references via assignment or transaction invocation. The compiler outputs an error if a reference to an `asset` goes out of scope while it is Owned. Ownership can be explicitly discarded with the `disown` operator.

Unowned is the complement to Owned: An object has at most one Owned reference but an arbitrary number of Unowned references. Unowned references are not linear, as they do not convey ownership. They are nonetheless useful. For example, a `Wallet` object might have owning references to `Money` objects, but a `Budget` object might have Unowned aliases to those objects so that the value of the `Money` can be tracked (even though only the `Wallet` is permitted to transfer the objects to another owner). Alternatively, if there is no owner of a non-asset object, then it may have Shared and Unowned aliases. Examples of some of these scenarios are shown in Figure 2 to provide some intuition. A summary of modes is shown in Table 2.

In Obsidian, the *mode* portion of a type can change due to operations on a reference, so transaction signatures can specify modes both before and after execution. As in Java, a first argument called `this` is optional; when present, it is used to specify initial and final modes on the receiver. The ≫ symbol separates the initial mode from the final one. In the example of Figure 1, the signature of buy (lines 27 and 28) indicates that buy must be invoked on a `TinyVendingMachine` that is statically known to be in state `Full`, passing a `Coin` object that the caller owns. When buy returns, the receiver will be in state `Empty` and the caller will no longer have ownership of the `Coin` argument.

Obsidian contracts can have constructors (line 12 above), which initialize fields as needed. If a contract has any states declared, then every instance of the contract must be in one of those states from the time each constructor exits.

Objects in smart contracts frequently maintain high-level state information [Ethereum Foundation 2020c], with the set of permitted transactions depending on the current state. For example, a `TinyVendingMachine` might be `Empty` or `Full`, and the buy transaction can only be invoked on a `Full` machine. Prior work showed that including state information in documentation helped users understand how to use object protocols [Sunshine et al. 2014], so we include first-class support for states in Obsidian. By using typestate in Obsidian, the compiler can ensure that objects are manipulated correctly according to their states. State information can be captured in a mode. For example, `TinyVendingMachine@Full` is the type of a reference to an object of contract `TinyVendingMachine` with mode `Full`. In this case, the mode denotes that the referenced object is statically known to be in state `Full`.

State is mutable; objects can transition from their current state to another state via a transition operation. For example, `->Full(inventory = c)` sets the state of a `TinyVendingMachine` to the `Full` state, initializing the `inventory` field of the `Full` state to `c`. This leads to a potential difficulty: What if a reference to a `TinyVendingMachine` with mode `Empty` exists while the state transitions to `Full`? To prevent this problem, typestate is only available with references that also have ownership. Because of this, there is no need to separately denote ownership in the syntax; we simply observe that every typestate-bearing reference is also owned. Then, Obsidian restricts the operations that can be performed through a reference according to the reference's mode. In particular, if an owned reference might exist, then non-owning references cannot be used to mutate typestate. If no owned references exist, then all references permit state mutation. In contrast, although an object may have multiple Shared aliases, those references do not specify typestate, and an object that has an Owned reference cannot also have a Shared reference, so the same soundness problem does not exist for Shared.

States can be defined with the `asset` keyword, in which case an instance of the contract represents an asset that should not be lost whenever it is in that state. For example, an insurance policy might own `Money` while the policy is active to ensure that claims can be paid, but after the policy changes to an expired state, the policy no longer holds the money.

As another example of the relevance of linearity in smart contracts, we wrote an Obsidian implementation of the ERC-20 token standard [Vogelsteller and Buterin 2015], which is shown below. In our implementation, the basic arithmetic of token value is implemented in `ExampleToken`, which is trusted code. In practice, this can be implemented in reusable library code. In contrast, the rest of the implementation in `ExampleTokenBank` does not use arithmetic directly. As a result, the compiler can ensure that no instances of `ExampleToken` are lost accidentally.

The example shows how linear assets are managed in Obsidian. In lines 84 and 92, the tokens owned by the *from* and *to* accounts are temporarily removed from the `balances` dictionary. Then, the appropriate amount of tokens are split off in line 101 and merged with the *to* account's tokens

on line 102. Finally, the tokens are restored to the `balances` dictionary. This style of manipulation is common in Obsidian: Rather than mutating the tokens inside a collection, they are removed and tracked linearly, and then the new balance is restored. In each operation, the type system tracks ownership of the tokens. Any arithmetic is encapsulated inside the token implementation. This contrasts with a typical Solidity implementation, in which arithmetic is used directly and is not checked by the compiler.

The example works within the framework of ERC-20. However, linearity allows another possible way to design token economies. Instead of having a central tracking system (a bank) on the blockchain recording who owns each number of tokens, we could regard token objects as *capabilities* [Boyland et al. 2001] that have value independent of a central bank. Then, no central contract on the blockchain need track a mapping from identity to balance; instead, those who own tokens will hold owning references to their tokens, and to transfer ownership, the owner can leverage the ownership transfer semantics in Obsidian. Of course, when ownership passes from the blockchain to outside the blockchain, or to contracts not written in Obsidian, there must be dynamic checks at the interface to ensure that untrusted code does not duplicate tokens.

The implementation leverages `Dict`, a polymorphic dictionary implementation. In practice, the implementation of `Dict` might be replaced by a platform-native primitive, such as native mappings on Ethereum or `HashMap` instances on Hyperledger Fabric.

```
1  import "Dict.obs"
2  import "Integer.obs"
3
4  asset interface ObsidianToken {
5    transaction getValue() returns int;
6    transaction merge(ObsidianToken@Owned >> Unowned other);
7    transaction split(int val) returns ObsidianToken@Owned;
8  }
9
10 asset contract ExampleToken implements ObsidianToken {
11   int value;
12
13   ExampleToken@Owned(int v) {
14     value = v;
15   }
16
17   transaction getValue(ExampleToken@Unowned this) returns int {
18     return value;
19   }
20
21   transaction merge(ObsidianToken@Owned >> Unowned other) {
22     value = value + other.getValue();
23     disown other;
24   }
25
26   transaction split(ExampleToken@Owned this, int val) returns ExampleToken@Owned {
27     if (val > value) {
28       revert ("Can't split off more than the existing value");
29     }
30     ExampleToken other = new ExampleToken(val);
31     value = value - val;
32     return other;
33   }
34 }
35
```

```
36  // ERC20 has been slightly adapted for Obsidian, since Obsidian does not have
37  //   a built-in authentication mechanism.
38  asset interface ERC20 {
39    transaction totalSupply() returns int;
40    transaction balanceOf(int ownerAddress) returns int;
41    transaction transfer(int fromAddress, int toAddress, int value) returns bool;
42
43    // - allow ownerAddress to withdraw from your account,
44    //   multiple times, up to the value amount.
45    transaction approve(int ownerAddress, int fromAddress, int value) returns bool;
46
47    // Returns the amount of allowance still available.
48    transaction allowance(int ownerAddress, int fromAddress) returns int;
49
50    // Transfers tokens from an allowance that has already been granted.
51    transaction transferFrom(int senderAddr, int fromAddr, int toAddr, int value)
52          returns bool;
53  }
54
55  main asset contract ExampleTokenBank implements ERC20 {
56    int totalSupply;
57    Dict[Integer, ExampleToken]@Owned balances;
58
59    // map from fromAddress to (map from spender to amount)
60    Dict[Integer, Dict[Integer, Integer]@Owned]@Owned allowed;
61
62    ExampleTokenBank@Owned() {
63      totalSupply = 0;
64      balances = new Dict[Integer, ExampleToken@Owned](new IntegerComparator());
65      allowed = new Dict[Integer, Dict[Integer, Integer]@Owned](new IntegerComparator());
66    }
67
68    transaction totalSupply() returns int {
69      return totalSupply;
70    }
71
72    transaction balanceOf(int ownerAddress) returns int {
73      Option[ExampleToken@Unowned] balance = balances.peek(new Integer(ownerAddress));
74      if (balance in None) {
75        return 0;
76      }
77      else {
78        return balance.unpack().getValue();
79      }
80    }
81
82    transaction transfer(int fromAddress, int toAddress, int value) returns bool {
83      Integer fromIntegerAddress = new Integer(fromAddress);
84      Option[ExampleToken@Owned] fromBalance = balances.remove(fromIntegerAddress);
85      if (fromBalance in None) {
86        return false;
87      }
88      else {
89        ExampleToken fromTokens = fromBalance.unpack();
90        if (value <= fromTokens.getValue()) {
91          Integer toIntegerAddress = new Integer(toAddress);
92          Option[ExampleToken@Owned] toBalance = balances.remove(toIntegerAddress);
93          ExampleToken toTokens;
94          if (toBalance in Some) {
95            toTokens = toBalance.unpack();
96          }
97          else {
98            toTokens = new ExampleToken(0); // 0 value
```

```
99          }
100
101         ExampleToken tokensToMove = fromTokens.split(value);
102         toTokens.merge(tokensToMove);
103         balances.insert(toIntegerAddress, toTokens);
104         balances.insert(fromIntegerAddress, fromTokens);
105
106         return true;
107       }
108       else {
109         // Insufficient funds available.
110         balances.insert(fromIntegerAddress, fromTokens);
111         return false;
112       }
113     }
114   }
115
116   // Records a new allowance. Replaces any previous allowance.
117   transaction approve(int ownerAddress, int fromAddress, int value) returns bool {
118     Integer ownerAddressInteger = new Integer(ownerAddress);
119     Option[Dict[Integer, Integer]@Owned] ownerAllowancesOption =
120         allowed.remove(ownerAddressInteger);
121
122     Dict[Integer, Integer] ownerAllowances;
123     if (ownerAllowancesOption in None) {
124       ownerAllowances = new Dict[Integer, Integer@Owned](new IntegerComparator());
125     }
126     else {
127       ownerAllowances = ownerAllowancesOption.unpack();
128     }
129
130     Option[Integer@Owned] oldAllowance = ownerAllowances.replace(
131                                             new Integer(fromAddress),
132                                             new Integer(value));
133     allowed.insert(ownerAddressInteger, ownerAllowances);
134
135     // Options are assets because they CAN hold assets,
136     // but this one doesn't happen to do so.
137     disown oldAllowance;
138     return true;
139   }
140
141   transaction allowance(int ownerAddress, int fromAddress) returns int {
142     Option[Dict[Integer, Integer]@Unowned] ownerAllowancesOption =
143         allowed.peek(new Integer(ownerAddress));
144     switch (ownerAllowancesOption) {
145       case None {
146         return 0;
147       }
148       case Some {
149         Dict[Integer, Integer@Owned] ownerAllowances = ownerAllowancesOption.unpack();
150         Option[Integer@Unowned] spenderAllowance =
151             ownerAllowances.peek(new Integer(fromAddress));
152         if (spenderAllowance in None) {
153           return 0;
154         }
155         else {
156           return spenderAllowance.unpack().getValue();
157         }
158       }
159     }
160   }
161
```

```
162     // senderAddress wants to transfer value tokens from fromAddress to toAddress.
163     // This requires that an allowance have been set up in advance and that
164     // fromAddress has enough tokens.
165     transaction transferFrom(int senderAddr, int fromAddr, int toAddr, int value)
166                 returns bool
167     {
168       int allowance = allowance(senderAddress, fromAddress);
169       if (allowance >= value) {
170         int newAllowance = allowance - value;
171         bool transferSucceeded = transfer(fromAddress, toAddress, value);
172         if (!transferSucceeded) {
173           // Perhaps not enough tokens were available to transfer.
174           return false;
175         }
176         approve(senderAddress, fromAddress, newAllowance);
177
178         return true;
179       }
180       else {
181         return false;
182       }
183     }
184   }
```

4 OBSIDIAN LANGUAGE DESIGN PROCESS AND DETAILS

Obsidian is the first object-oriented language (of which we are aware) to integrate linear assets and typestate. This combination—and, in fact, even just including typestate—could result in a design that was hard to use, since typical typestate languages require users to understand a complex permissions model. We also aimed to simplify the job of the programmer relative to existing blockchain programming languages by eliminating onerous, error-prone programming tasks, such as writing serialization and deserialization code. In this section, we describe how we designed language features to improve user experience, in some cases driven by results of formative user studies [Barnaby et al. 2017]. Some other system features, such as serialization, are discussed in Section 6. Rather than relying only on our own experience and intuition, we invited participants to help us assess the tradeoffs of different design options. This enabled us to take a more data-driven approach in our language design, as suggested by Stefik and Hanenberg [2014] and Coblenz et al. [2018]. We take the perspective that we should integrate *qualitative* methods in addition to quantitative methods to drive language design in a direction that is more likely to be beneficial for users.

The syntax of Obsidian is specified in Section 7 as an extension and modification of the syntax of Silica, and the semantics are defined by translation to Silica.

4.1 Type Declarations, Annotations, and Static Assertions

Obsidian requires type declarations of local variables, fields, and transaction parameters. In addition to providing familiarity to programmers who have experience with other object-oriented languages, there is a hypothesis that these declarations may aid in usability by providing documentation, particularly at interfaces [Coblenz et al. 2014]. Traditional declarations are also typical in prior typestate-supporting languages, such as Plaid [Sunshine et al. 2011]. Unfortunately, typestate is incompatible with the traditional semantics of type declarations: programmers normally expect that the type of a variable always matches its declared type, but mutation can result in the typestate no longer matching the initial type of an identifier. This violates the *consistency* usability heuristic [Nielsen and Molich 1990] and is a potential source of reduced code readability, since

determining the type of an identifier can require reading all the code from the declaration to the program point of interest.

To alleviate this problem, we introduced *static assertions*. These have the syntax `[e @ mode]`. For example, `[account @ Open]` statically asserts that the reference `account` is owned and refers to an object that the compiler can prove is in `Open` state. Furthermore, to avoid confusion about the meanings of local variable declarations, Obsidian forbids mode specifications on local variable declarations.

Static assertions have no implications on the dynamic semantics (and therefore have no execution time cost); instead, they serve as checked documentation. The type checker verifies that the given mode is valid for the expression in the place where the assertion is written. A reader of a typechecked program can be assured, then, that the specified types are correct, and the author can insert the assertions as needed to improve program understandability.

4.2 State Transitions

Each state definition can include a list of fields, which are in scope only when the object is in the corresponding state (see line 8 of Figure 1). What, then, should be the syntax for initializing those fields when transitioning to a different state? Some design objectives included:

- When an object is in a particular state, the fields for that state should be initialized.
- When an object is *not* in a particular state, the fields for that state should be out of scope.
- According to the *user control and freedom* heuristic [Nielsen and Molich 1990] and results by Stylos et al. [Stylos and Clarke 2007], programmers should be able to initialize the fields in any order, including by assignment. Under this criterion, it does not suffice to only permit constructor-style simultaneous initialization.

To allow maximum user flexibility without compromising the integrity of the type system, we implemented a flexible approach. When a state transition occurs, all fields of the target state must be initialized. However, they can be initialized either *in* the transition (e.g., `->S(x = a)` initializes the field `x` to `a`) or *prior to* the transition (e.g., `S::x = a; ->S`). In addition, fields that are in scope in the current state but will not be in scope in the target state must *not* be owned references to assets at the transition. Ownership of fields that will go out of scope in a transition must first be transferred to another reference or disowned before the transition.

4.3 Transaction Scope

Transactions in Obsidian are invoked on a particular receiving object, and are only available when the receiver is in a particular state. Correspondingly, other typestate-oriented languages support defining methods inside states. For example, Plaid [Sunshine et al. 2011] allows users to define the `read` method inside the `OpenFile` state to make clear that `read` can only be invoked when a `File` is in the `OpenFile` state. However, this is problematic when methods can be invoked when the object is in several states.

Barnaby et al. [2017] considered this question for Obsidian and observed that study participants, who were given a typestate-oriented language that included methods in states, asked many questions about what could happen during and after state transitions. They were unsure what `this` meant in that context and what variables were in scope at any given time. One participant thought it should be disallowed to call transactions available in state `S1` while writing a transaction that was lexically in state `Start`. For this reason, we designed Obsidian so that transactions are defined lexically *outside* states. Transaction signatures indicate (via type annotations on a first argument called `this`) from which states each transaction can be invoked. This approach is consistent with other languages, such as Java, which also allows type annotations on a first argument `this`.

```
1  contract Wallet {
2    Money@Owned money;
3
4    transaction swap (Money @ Owned m) returns Money @ Owned {
5      Money result = money;
6      money = m;
7      return result;
8    }
9  }
```

Fig. 3. Obsidian's approach for handling transitions.

```
1  contract Wallet {
2    state Empty;
3    state Full {
4      Money @ Owned money;
5    }
6
7    transaction swap (Wallet@Full this, Money @ Owned m)
8                     returns Money @ Owned
9    {
10     // Suppose the transition returns the contents of the old field.
11     Money result = ->Empty;
12     ->Full(money = m);
13     return result;
14   }
15 }
```

Fig. 4. An alternative approach for handling transitions.

4.4 Field Type Consistency and Private Transactions

In traditional object-oriented languages, fields always refer either to `null` or to objects whose types are subtypes of the fields' declared types. This presents a difficulty for Obsidian, since the mode is part of the type, and the mode can change with operations. For example, a `Wallet` might have a reference of type `Money@Owned`. How should a programmer implement `swap`? One way is shown in Figure 3.

The problem is that line 5 changes the type of the `money` field from `Owned` to `Unowned` by transferring ownership to `result`. Should this be a type error, since it is inconsistent with the declaration of `money`? If it is a type error, then how is the programmer supposed to implement `swap`? One possibility is to add another state, as shown in Figure 4.

Although this approach might seem like a reasonable consequence of the desire to keep field values consistent with their types, it imposes a significant burden. First, the programmer is required to introduce additional states, which leaks implementation details into the interface (unless we mitigate this problem by making the language more complex, e.g., with `private` states or via abstraction over states). Second, this requires that transitions return the newly out-of-scope fields, but it is not clear how: Should the result be of record type? Should it be a tuple? What if the programmer neglects to do something with the result? Plaid [Sunshine et al. 2011] addressed the problem by not including type names in fields, but that approach may hamper code understandability [Coblenz et al. 2014].

In Obsidian, we permit fields to *temporarily* reference objects that are not consistent with the fields' declarations, but we require that at the end of transactions (and constructors), the fields refer to appropriately typed objects. This approach is consistent with the approach for local variables, with the additional postcondition of type consistency. Both local variables and fields of nonprimitive type, and transaction parameters must always refer to instances of appropriate contracts; the only discrepancy permitted is of mode. Obsidian forbids re-assigning formal parameters to refer to other objects to ensure soundness of this analysis.

This design decision introduces a problem with re-entrancy: Re-entrant calls from the middle of a transaction's body, where the fields may not be consistent with their types, can be dangerous, since the called transactions are supposed to be allowed to assume that the fields reference objects consistent with the fields' types. One way to address this would be by forbidding all re-entrant calls at an object level of granularity (i.e., only one transaction with a given receiver can be on the call stack at a given time). However, we regard this as too restrictive, as it precludes even writing helper transactions.

Instead, Obsidian distinguishes between *public* and *private* transactions. To facilitate refactoring code into appropriate transactions, Obsidian allows private transactions that can be invoked when fields have types that are inconsistent with their declarations. To enable this, `private` transactions declare the expected types of the fields before and after the invocation. For example:

```
contract AContract {
    state S1;
    state S2;

    AContract@S1 c;
    private (AContract@S2 >> S1 c) transaction t1() {...}
}
```

Transaction `t1` may only be invoked by transactions of `AContract`, only on `this`, and only when `this.c` temporarily has type `AContract@S2`. When `t1` is invoked, the compiler checks to make sure field `c` has type `C@S2`, and assumes that after `t1` returns, `c` will have type `AContract@S1`. Of course, the body of `t1` is checked assuming that `c` has type `C@S2` to make sure that afterward, `c` has type `C@S1`.

This approach allows programmers to extract portions of their transactions into private transactions, which have specified pre- and post- conditions regarding the field types. The restriction that these transactions are private arises, because the type checker only tracks the types of fields of `this` individually (and assumes all other objects have fields of types consistent with their declarations).

Avoiding unsafe re-entrancy has been shown to be important for real-world smart contract security, as millions of dollars were stolen in the DAO hack via a re-entrant call exploit [Daian 2016].

4.5 Dynamic State Checks

The Obsidian compiler enforces that transactions can only be invoked when it can prove statically that the objects are in appropriate states according to the signature of the transaction to be invoked. In some cases, however, it is impossible to determine this statically. For example, consider `redeem` in Figure 5. At the beginning of the transaction, the contract may be in either state `Active` or state `Expired`. However, inside the dynamic state check block that starts on line 29, the compiler assumes that `this` is in state `Active`. The compiler generates a dynamic check of state according to the test. However, regarding the code in the block, there are two cases. If the dynamic state

```
1  main asset contract GiftCertificate {
2      Date @ Unowned expirationDate;
3
4      state Active {
5          Money @ Owned balance;
6      }
7
8      state Expired;
9      state Redeemed;
10
11     GiftCertificate(Money @ Owned >> Unowned b, Date @ Unowned d)
12     {
13         expirationDate = d;
14         ->Active(balance = b);
15     }
16
17     transaction checkExpiration(GiftCertificate @ Active >> (Active | Expired) this)
18     {
19         if (getCurrentDate().greaterThan(expirationDate)) {
20             disown balance;
21             ->Expired;
22         }
23     }
24     transaction redeem(GiftCertificate @ Active >> (Expired | Redeemed) this)
25                 returns Money@Owned
26     {
27         checkExpiration();
28
29         if (this in Active) {
30             Money result = balance;
31             ->Redeemed;
32             return result;
33         }
34         else {
35             revert "Can't redeem expired certificate";
36         }
37     }
38     transaction getCurrentDate(GiftCertificate @ Unowned this)
39                 returns Date @ Unowned
40     {
41         return new Date();
42     }
43 }
```

Fig. 5. A dynamic state check example.

check is of an Owned reference x, then it suffices for the type checker to check the block under the assumption that the reference is of type according to the dynamic state check. However, if the reference is Shared, then there is a problem: What if code in the block changes the state of the object referenced by x? This would violate the expectations of the code inside the block, which is checked as if it had ownership of x. We consider the cases, since the compiler always knows whether an expression is Owned, Unowned, or Shared:

- If the expression to be tested is a variable with Owned mode, then the body of the if statement can be checked assuming that the variable initially references an object in the specified state, since that code will only execute if that is the case due to the dynamic check.

- If the expression to be tested is a variable with Unowned mode, then there may be another owner (and the variable cannot be used to change the state of the referenced object anyway). In that case, typechecking of the body of the `if` proceeds as if there had been no state test, since it would be unsafe to assume that the reference is owned. However, this kind of test can be useful if the desired behavior does not statically require that the object is in the given state. For example, in a university accounting system, if a `Student` is in `Enrolled` state, then their account should be debited by the cost of tuition this semester. The `debit` operation does not directly depend on the student's state; the state check is a matter of policy regarding who gets charged tuition.
- If the expression to be tested is a variable with Shared mode, then the runtime maintains a state lock that pertains to other shared references. The body is checked initially assuming that the variable owns a reference to an object in the specified state. Then, the type checker verifies that the variable still holds ownership at the end and that the variable has not been re-assigned in the body. However, at execution time, if any *other* Shared reference is used to change the state of the referenced object (for example, via another alias used in a transaction that is invoked by the body of the dynamic state check block), then the transaction is aborted (recall that the blockchain environment is sequential, so there is only one top-level transaction in progress at a time). This approach enables safe code to complete but ensures that the analysis of the type checker regarding the state of the referenced object remains sound. This approach also bears low execution time cost, since the cost of the check is borne only in transitions via Shared references. An alternative design would require checks at invocations to make sure that the referenced object was indeed in the state the type checker expected, but we expect our approach has significantly lower execution time cost. Furthermore, our approach results in errors occurring immediately on transition. The alternative approach would give errors only when the referenced object was used, which could be substantially after the infringing transition, which would require the programmer to figure out which transition caused the bug.
- If the expression to be tested is not a variable, then the body of the `if` statement is checked in the same static context as the if statement itself. It would be unsafe for the compiler to make any assumptions about the type of future executions of the expression, since the type may change. This case only occurs in Obsidian, not in the underlying Silica formalism, which is in A-normal form [Sabry and Felleisen 1992].

The dynamic state check mechanism is related to the *focusing* mechanism of Fahndrich and DeLine [2002]. Dynamic state checks in Obsidian detect unsafe uses of aliases more precisely (less conservatively) than focusing, enabling many more safe programs to typecheck. Furthermore, Obsidian does not require the programmer to specify *guards*, which in focusing enable the compiler to reason conservatively about which references may alias.

4.6 Parametric Polymorphism

Parametric polymorphism is particularly important for Obsidian to maintain safety of collections and avoid needless code duplication. Requiring users to cast objects retrieved from containers to the appropriate type would defeat the point of the language, which is to provide strong static guarantees, since those casts would have to be checked dynamically. Furthermore, there would have to be separate containers for different modes, since a container's elements would need to be either Unowned, Shared, or Owned. In Obsidian, a contract can have *two* type parameters: one for a contract and one for a mode. For example, part of the polymorphic LinkedList implementation is as follows:

```
1  contract LinkedList[T@s] {
2      state Empty;
3      state HasNext {
4          LinkedList[T@s]@Owned next;
5          T@s value;
6      }
7      transaction append(LinkedList@Owned this, T@s >> Unowned obj) {
8        ...
9    }
10 }
```

Line 1 shows that the contract type is parameterized by the contract variable `T`, and the mode is parameterized by the mode variable `s`. In line 4, the `next` field is an Owned reference to an object of type `LinkedList[T@s]` – that is, a node whose type parameters are the same as the containing contract's type parameters. An object of type `LinkedList[Money@Owned]` is a container that holds a list of `Money` references, each of which the container owns. Using a separate parameter for the mode allows parameterization over states, e.g., a `LinkedList[LightSwitch@On]` owns references to `LightSwitch` objects that are each in the `On` state. In line 7, appending an element to a LinkedList always takes any ownership that was given, and the parameter `obj` must conform to the type specified by the type parameter `T@s`.

5 SYSTEM DESIGN AND IMPLEMENTATION

Our current implementation of Obsidian supports Hyperledger Fabric [The Linux Foundation 2020], a permissioned blockchain platform. In contrast to public platforms, such as Ethereum, Fabric permits organizations to decide who has access to the ledger, and which peers need to approve (*endorse*) each transaction. This typically provides higher throughput and more convenient control over confidential data than public blockchains, allowing operators to trade off distributed trust against high performance. Fabric supports smart contracts implemented in Java, so the Obsidian compiler translates Obsidian source code to Java for deployment on Fabric peer nodes. The Obsidian compiler prepares appropriately structured directories with Java code and a build file. Fabric builds and executes the Java code inside purpose-build Docker containers that run on the peer nodes. The overall Obsidian compiler architecture is shown in Figure 6.

The type checker is syntax-directed and therefore relatively cheap to run. As is typical, the type checker maintains a context mapping variables to types as it iterates through the body of each transaction, and updates the context with the changes that result from executing each statement. Since local variable declarations do not include modes, local variables start out with a specified context and "inferred" mode; the mode is updated as soon as an assignment occurs. Otherwise, variables always have a known mode; after branches, the output contexts of both branches are merged as is specified in the Silica static semantics (AJ:33 in Appendix A). If the merge is not possible, then the compiler reports an error.

5.1 Storage in the Ledger

Fabric provides a key/value store for persisting the state of smart contracts in the ledger. As a result, Fabric requires that smart contracts serialize their state in terms of key/value pairs. In other smart contract languages, programmers are required to manually write code to serialize and deserialize their smart contract data. In contrast, Obsidian automatically generates serialization code, leveraging *protocol buffers* [Google Inc. 2019] to map between message formats and sequences of bytes. When a transaction is executed, the appropriate objects are lazily loaded from the key/value store as required for the transaction's execution. Lazy loading is *shallow*: The object's fields are

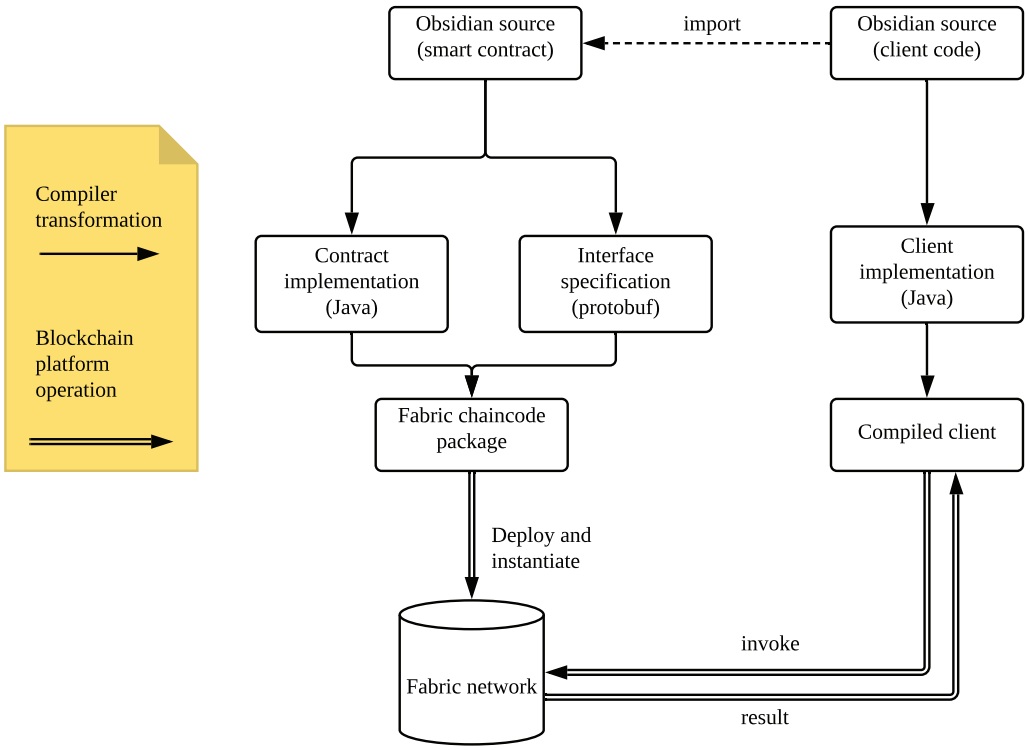

Fig. 6. Obsidian system architecture.

loaded, but objects that fields reference are not loaded until *their* fields are needed. After executing the transaction, Obsidian's runtime environment automatically serializes the modified objects and saves them in the ledger. This means that aborting a transaction and reverting any changes is very cheap, since this entails *not* setting key/value pairs in the store, flushing the heap of objects that have been lazily loaded, and (shallowly) re-loading the root object from the ledger. This lazy approach decreases execution cost and frees the programmer from needing to manually load and unload key/value pairs from the ledger, as would normally be required on Fabric.

Although transactions can invoke other transactions, the transactional semantics do not nest. For example, if T1 invokes T2, and T2 reverts, then any changes made by T1 are also reverted.

5.2 Obsidian Client Programs

The convention for most blockchain systems is that smart contracts are written in one language, such as Solidity, and client programs are written in a different language, such as JavaScript. Unfortunately, in Solidity, transaction arguments and outputs must be primitives, not objects; arrays of bytes can be transferred, but the client and server must each implement corresponding serialization and deserialization code. The interface for a given contract is specified in an Application Binary Interface (ABI), documented in a schema written in JavaScript. If there are any incompatibilities between the semantics of the JavaScript serialization code and the semantics of the Solidity contract that interprets the serialized message, then there can be bugs.

Obsidian addresses this problem by allowing users to write client programs in Obsidian. Client programs can reference the same contract implementations that were instantiated on the server, obviating the need for two different implementations of data structures. Clients use the same

```
import "TinyVendingMachine.obs"

main contract TinyVendingMachineClient {
    transaction main(remote TinyVendingMachine@Shared machine) {
        restock(machine);

        if (machine in Full) {
            Coin c = new Coin();
            remote Candy candy = machine.buy(c);
            eat(candy);
        }
    }

    private transaction restock(remote TinyVendingMachine@Shared machine) {
        if (machine in Empty) {
            Candy candy = new Candy();
            machine.restock(candy);
        }
    }

    private transaction eat(remote Candy @ Owned >> Unowned c) {
        disown c;
    }
}
```

Fig. 7. A simple client program, showing how clients reference a smart contract on the blockchain. Note that the blockchain-side smart contract has been modified (relative to Figure 1) to have Shared receivers, since top-level objects are never owned by clients.

automatically generated serialization and deserialization code that the server does. As a result, Obsidian permits arbitrary objects (encoded via protocol buffers) to be passed as arguments and returned from transactions. Since the protocol buffer specifications are emitted by the Obsidian compiler, any client (even non-Obsidian clients) can use these specifications to correctly serialize and deserialize native Obsidian objects to invoke Obsidian transactions and interpret their results.

The Obsidian client program has a `main` transaction, which takes a `remote` reference. The keyword `remote`, which modifies types of object references, indicates that the type refers to a remote object. The compiler implements remote references with stubs, via an RMI-like mechanism. When a non-`remote` reference is passed as an argument to a remote transaction, the referenced object is serialized and sent to the blockchain. Afterward, the reference becomes a `remote` reference, so that only one copy of the object exists (otherwise mutations to the referenced object on the client would not be reflected on the blockchain, resulting in potential bugs). This change in type is similar to how reference modes change during execution. Figure 7 shows a simple client program that uses the `TinyVendingMachine` above. The `main` transaction takes a `remote` reference to the smart contract instance.

Every Obsidian object has a unique ID, and references to objects can be transmitted between clients and the blockchain via object ID. There is some subtlety in the ID system in Obsidian: All blockchain transactions must be deterministic so that all peers generate the same IDs, so it is impossible to use traditional (e.g., timestamp-based or hardware-based) UUID generation. Instead, Obsidian bases IDs on transaction identifiers, which Fabric provides, and on an index kept in an ID factory. Since transaction IDs are unique, each transaction can have its own ID factory and still avoid collisions. The initial index is reset to zero at the beginning of each transaction so that no

state pertaining to ID generation needs to be stored between transactions. Blockchains provide a sequential execution environment, so there is no need to address race conditions in ID generation. When clients instantiate contracts, they generate IDs with a traditional UUID algorithm, since clients operate off the blockchain.

Blockchains allow clients to interleave their transactions arbitrarily. This does not suffice to ensure safety in arbitrary Obsidian client programs. For example, suppose a client has code like this, assuming that `c` is a reference to a remote (on-blockchain) object:

```
1  if (c in S1) {
2          c.t1(); // suppose t1 requires that c references an object in state S1
3  }
```

In the current implementation, because the test and the t1() invocation execute remotely, it is possible that an intervening transaction could change the state of the referenced object so that it is no longer in state S1. In the future, however, Obsidian will address this issue in a platform-appropriate manner. Once the programmer identifies a critical section, one approach is for the client to wrap the section in a lambda so that the server can execute it in one transaction. This approach might work well on Ethereum, where clients must pay for the costs of executing code on the blockchain. However, on Fabric, this approach is problematic, because the security policy is such that clients should not force the blockchain to execute arbitrary code (for example, including non-terminating code). An approach that may be more effective is to use *optimistic concurrency* [Kung and Robinson 1981], in which smart contracts on the blockchain defer commitment of changes from clients until the client's critical section is done; then, either the transaction is committed, or the changes are discarded because of intervening changes that occurred.

5.3 Ensuring Safety with Untrusted Clients

If a client program is written in a language other than Obsidian, then it may not adhere to Obsidian's type system. For example, a client program may obtain an owned reference to an object and then attempt to transfer ownership of that object to multiple references on the blockchain. This is called the *double-spend* problem on blockchains: A program may attempt to consume a resource more than once. To address this problem, the Obsidian runtime keeps a list of all objects for which ownership has been passed outside the blockchain. When a transaction is invoked on an argument that must be owned, the runtime aborts the transaction if that object is not owned outside the blockchain and otherwise removes the object from the list. Likewise, when a transaction argument or result becomes owned by the client after the transaction (according to the transaction's signature), the runtime adds the object to the list. Of course, Obsidian has no way of ensuring safe manipulation of owned references in non-Obsidian clients, but this approach ensures that each time an owned reference leaves the blockchain, it only returns once, preventing double-spending attacks. Obsidian cannot ensure that non-Obsidian clients do not lose their owned references, so we hope that most client code that manipulates assets will be written in Obsidian.

6 SILICA

In this section, we describe Silica, the core calculus that forms a foundation for Obsidian. Silica is so named because obsidian glass is composed in large part of silica (65% to 80%) [Encyclopædia Britannica 2020]. Silica is designed in the style of Featherweight Typestate [Garcia et al. 2014], which is itself designed in the style of Featherweight Java [Igarashi et al. 2001]. Silica leverages key concepts and notation, such as *type splitting*, from Featherweight Typestate. However, Silica differs significantly from FT; the differences are described in more detail in Section 2.3.

Figure 8 shows the syntax of Silica. Silica uses A-normal form [Sabry and Felleisen 1992] as a simplification to avoid nested expressions in most cases. Note that `[]` in typewriter typeface indicates a static assertion, whereas Roman [] indicates an optional part of the syntax. Following Featherweight Java [Igarashi et al. 2001], the syntax specification uses a horizontal line above a symbol to indicate that it is a list. To distinguish these lines from the lines that denote judgments, sequences will be denoted with a thick, orange line, whereas judgments will use a thin, black line.

T represents the type of a reference to an object. It is divided into two parts: T_C represents the contract (analogous to a class) and T_{ST} represents the *mode*. A contract is either a declared type with some (possibly zero) type parameters, or a type variable. A mode reflects any ownership held by the reference (a *permission*) and any state specification.

Contracts are defined in terms of type parameters T_G, states ST, and transactions M. Constructors for a contract C return a reference with permission P. Each argument is passed with initial type T, but because the constructor assigns parameters to fields, ownership of parameters may be consumed, resulting in a new mode T_{ST}. Changes to types are denoted with $T \gg T_{ST}$. States ST consist of a state name S and a collection of fields $\overline{F}$. In this syntax, if a name is re-used in different states, then the field is in scope in all of those states. In the implementation, rather than re-using names, there is special syntax for this: `T f available in S1, S1` declares field `f` that is in scope in states `S1` and `S2`. This approach avoids duplicated code in user programs.

Because Silica is an expression language, not a statement language, any sequencing must occur via nested let-bindings. Here, s denotes a *simple expression*. For now, simple expressions are merely variables; we will see in the dynamic semantics that sometimes simple expressions can represent *indirect references* to memory locations.

Transaction signatures specify initial and final modes for `this` and for the parameters. Signatures of private transactions also specify initial and final modes for the fields, which for private transactions may differ from their declared modes.

6.1 Silica Static Semantics

The static semantics make use of auxiliary judgments, which are defined in Appendix A. The auxiliary judgments are numbered and referenced by number, as in *AJ:22*.

6.1.1 *Preliminary Judgments.*

Typing contexts Δ and **type bound contexts** Γ

The typing context Δ includes local variables as well as temporary field types, which allow fields to temporarily have modes that differ from those in the fields' declarations. Type bounds Γ is a set of generic type variables $\mathrm{T_G}$ as defined in the grammar in Figure 8. Γ is used to track the typing constraints on type variables.

It is assumed that Δ and Γ are permuted as needed to apply the rules, but when a context is extended with a mapping, the new mapping replaces any previous mapping of the same variable.

$$
\begin{array}{lcl}
\Gamma & ::= & \cdot \\
 & | & \Gamma, \mathrm{T_G} \\
\Delta & ::= & \cdot \\
 & | & \Delta, \mathrm{x} : \mathrm{T} \\
 & | & \Delta, \mathrm{s.f} : \mathrm{T}
\end{array}
$$

$T_1 \Rrightarrow T_2/T_3$ **Type splitting**

Type splitting specifies how ownership of objects can be shared among aliases. In $T_1 \Rrightarrow T_2/T_3$, there is initially one reference of type T_1; afterward, there are two references of type T_2 and T_3. For example, an Owned reference can be split into an Owned reference and an Unowned reference.

$C \in$ ContractNames
$I \in$ InterfaceNames
$D \in$ ContractNames $\cup$ InterfaceNames
$X \in$ DeclarationVariables
$x \in$ IdentifierNames
$m \in$ transactionNames
$S \in$ StateNames
$p \in$ PermissionVariables
$f \in$ FieldNames

T	::=	$T_C@T_{ST}$	(types of contract references)
	| ()		
T_C	::=	$D\langle\overline{T}\rangle$	(types of contracts/interfaces)
	|	X	(declaration variables)
T_{ST}	::=	$\overline{\text{S}}$	(state disjunction)
	|	p	(permission/state variables)
	|	P	(concrete permission)
P	::=	Owned | Unowned | Shared	
T_G	::=	[asset] X@p implements $I\langle\overline{T}\rangle@T_{ST}$	(generic type parameter)
CON	::=	contract $C\langle\overline{T_G}\rangle$ implements $I\langle\overline{T}\rangle$ $\{\overline{ST}\ \overline{\text{M}}\}$	
$IFACE$	::=	interface $I\langle\overline{T_G}\rangle$ { $\overline{\text{ST}}$ $\overline{M_{SIG}}$ }	
ST	::=	[asset] S $\overline{F}$	
F	::=	T f	
M_{SIG}	::=	T $m\langle\overline{T_G}\rangle$ $(\overline{T \gg T_{ST}}\ x)$ $T_{ST} \gg T_{ST}$	(transaction specifying types for return, arguments, and receiver)
	|	private $\overline{T_{ST} \gg T_{ST}\ f}$ T $m\langle\overline{T_G}\rangle(\overline{T \gg T_{ST}\ x})$ $\text{T}_{\text{ST}} \gg \text{T}_{\text{ST}}$	(private transactions also specify field types)
M	::=	M_{SIG} { return e }	
e	::=	s	
	|	s.f	(field access)
	|	$\text{s}.m\langle\overline{T}\rangle(\overline{\text{x}})$	
	|	let $x : T = e$ in e	
	|	new $C\langle\overline{T}\rangle@\text{S}(\overline{\text{s}})$	(contract fields, then state fields)
	|	$\text{s} \rightarrow_{\text{Owned\|Shared}} \text{S}(\overline{\text{s}})$	(State transition initializing fields)
	|	s.f := s	(field update, with 1-based indexing)
	|	s := s	(variable assignment)
	|	[s @ T_{ST}]	(static assert)
	|	if s in$_P$ T_{ST} then e else e	(state test, owned or shared s)
	|	disown s	(drop ownership of owned ref.)
	|	pack	
s	::=	x	(simple expressions)

Fig. 8. Abstract syntax of Silica.

The relation is defined such that if $T_1 \Rrightarrow T_2/T_3$, and one of the right-side types holds ownership, then ownership is held by T_2, not T_3. The *maybeOwned* judgment (AJ:22) checks a type to see if it could be owned; ownership is always known except when a type includes a type variable, in which case the judgment conservatively assumes that the type might be owned. The *nonAsset* judgment is defined in AJ:19. The *contract* function, defined in AJ:4, extracts contract names from types.

$$\frac{T_C = contract(T)}{T \Rrightarrow T/T_C@\mathsf{Unowned}}\ \textsc{Split-Unowned}$$

$$\frac{}{T_C@\mathsf{Shared} \Rrightarrow T_C@\mathsf{Shared}/T_C@\mathsf{Shared}}\ \textsc{Split-shared}$$

$$\frac{\Gamma \vdash nonAsset(T_C@T_{ST}) \qquad maybeOwned(T_C@T_{ST})}{T_C@T_{ST} \Rrightarrow T_C@\mathsf{Shared}/T_C@\mathsf{Shared}}\ \textsc{Split-owned-shared}$$

$$\frac{}{\mathsf{unit} \Rrightarrow \mathsf{unit}/\mathsf{unit}}\ \textsc{Split-unit}$$

6.1.2 Main Typing Judgments.

$\boxed{\Gamma;\Delta \vdash_s e : T \dashv \Delta'}$ **Well-typed expressions**

Unlike some traditional typing judgments, in addition to an *input* typing context Δ, Silica's typing judgment includes an *output* typing context Δ'. This is because an expression can change the mode of object references. For example, using a variable that references an object may consume ownership of the object.

Note that $\overline{e} : \overline{T}$ is defined to mean a sequence $\overline{e : T}$. Expressions are typechecked in the context of a simple expression s, which represents this. Initial programs are written using this, but evaluation of invocations will substitute indirect references for instances of this. The subscript on the turnstile tracks the value of this in the current invocation.

T-lookup relies on the split judgment ($T_1 \Rrightarrow T_2/T_3$) (6.1.1), which describes how a permission in T_1 can be split between T_2 and T_3. If ownership is retained after the split, then T_2 holds ownership and T_3 does not. Thus, T_2 will be the type of the present usage of s', whereas any future uses are left with the permission in T_3,

$$\frac{T_1 \Rrightarrow T_2/T_3}{\Gamma;\Delta, s' : T_1 \vdash_s s' : T_2 \dashv \Delta, s' : T_3}\ \textsc{T-lookup.}$$

In a let expression, the bound variable can be an owning reference to an asset, but if so, e_2 must consume the ownership (as indicated by *disposable*, AJ:20).

$$\frac{\Gamma;\Delta \vdash_s e_1 : T_1 \dashv \Delta' \qquad \Gamma;\Delta', x : T_1 \vdash_s e_2 : T_2 \dashv \Delta'', x : T_1' \qquad \Gamma \vdash disposable(T_1')}{\Gamma;\Delta \vdash_s \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta''}\ \textsc{T-let.}$$

In assignment operations, any ownership is transferred to the variable that is being written. This requires that the left-hand-side variable be of disposable type,

$$\frac{T_{s''} \Rrightarrow T^*/T^{**} \qquad \Gamma \vdash disposable(T_s')}{\Gamma;\Delta, s' : T_{s'}, s'' : T_{s''} \vdash_s s' := s'' : \mathsf{unit} \dashv \Delta'', s' : T^*, s'' : T^{**}}\ \textsc{T-assign.}$$

To check a new invocation, T-new checks that the types of the arguments are appropriate for the field declarations in the contract, considering the state in which the object is being initialized. The *subsOk* judgment (AJ:29), which is used in T-new, ensures that the given type parameters are

suitable according to the declaration of C. *stateFields* (AJ:1) is used to look up the types of the fields to which the parameters will be assigned,

$$\frac{\Gamma;\Delta \vdash_s \overline{s'} : \overline{T_{s'}} \dashv \Delta' \qquad \overline{\Gamma \vdash T_{s'} <: \mathit{stateFields}(C\langle\overline{T}\rangle, S)} \qquad \overline{\mathit{subsOk}_\Gamma(T, T_G)} \quad \mathit{def}(C) = \text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T_I}\rangle\ \{\ldots\}}{\Gamma;\Delta \vdash_s \text{new } C\langle\overline{T}\rangle@S(\overline{s'}) : C\langle\overline{T}\rangle@S \dashv \Delta'} \text{ T-NEW.}$$

When accessing a field of this (note that the s in the expression is identical to the s subscript in the judgement), there are two cases. In the first case (T-THIS-FIELD-DEF), the type of the field is consistent with the declared type of the field, in which case T-THIS-FIELD-DEF makes sure that the field is in scope in all possible current states of the referenced object (via *intersectFields*, AJ:3). In the second case (T-THIS-FIELD-CTXT), the field type has been updated due to an assignment, so the field type comes from an override in the context. In both cases, any ownership that was present is consumed from the field using type splitting,

$$\frac{s.f \notin \mathit{Dom}(\Delta) \qquad T_1\ f \in \mathit{intersectFields}(T) \qquad T_1 \Rrightarrow T_2/T_3}{\Gamma;\Delta, s : T \vdash_s s.f : T_2 \dashv \Delta, s : T, s.f : T_3} \text{ T-THIS-FIELD-DEF,}$$

$$\frac{T_1 \Rrightarrow T_2/T_3}{\Gamma;\Delta, s : T, s.f : T_1 \vdash_s s.f : T_2 \dashv \Delta, s : T, s.f : T_3} \text{ T-THIS-FIELD-CTXT.}$$

A field can be overwritten only if the current reference is disposable, since otherwise assignment might overwrite owning references to assets,

$$\frac{\Gamma;\Delta \vdash_s s.f : T_C@T_{ST} \dashv \Delta' \qquad \Gamma;\Delta' \vdash_s s_f : T_C@T'_{ST} \dashv \Delta'' \quad \Gamma \vdash \mathit{disposable}(T_C@T_{ST})}{\Gamma;\Delta \vdash_s s.f := s_f : \text{unit} \dashv \Delta'', s.f : T_C@T'_{ST}} \text{ T-FIELDUPDATE.}$$

T-INV defines typing when invoking a method m on an expression s_1 ($s_1.m\langle\overline{T_M}\rangle(\overline{s_2})$). In invocations (of both public and private transactions), if the type of an argument differs from the declared type of the formal parameter, then the final type of the argument may differ from the declared final type of the parameter. For example, passing an Owned argument as input to a parameter that expects an Unowned reference is allowed, but the caller retains ownership. The function *funcArg* (AJ:34) computes the resulting final types of the arguments given the types of the input arguments, since if an Owned reference is passed to an initially Unowned formal parameter, the caller retains ownership even though the declared final type in the formal parameter would be Unowned.

To check an invocation, T-INV first looks up the bound on the type of the receiver, since the receiver's type may include type variables ($\Gamma \vdash \mathit{bound}(T_C@T_{ST})$ is defined in AJ:23). Next, T-INV uses the bound and the declared type arguments in the invocation to specialize the types of the method with $\mathit{specializeTrans}_\Gamma\left(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle\right)$ (AJ:32). Then, T-INV checks to make sure the receiver is of appropriate type. T-INV checks to make sure that all the arguments are of appropriate type and that the fields are of types consistent with their declarations ($\forall f, s.f \notin \Delta$). The resulting types of the receiver and arguments are computed according to their initial types and the declaration of

the method,

$$\frac{\begin{array}{c}\Gamma \vdash bound(T_C@T'_{STs1}) = D\langle\overline{T}\rangle@T_{STs1} \\ specializeTrans_\Gamma(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle) = T\ m\langle\overline{T'_M}\rangle(\overline{T_{C_x}@T_x \gg T_{xST}\ x})\ T_{this} \gg T'_{this} e \\ \Gamma \vdash T_{STs1} <:_* T_{this} \qquad \overline{\Gamma \vdash T_{s2} <: T_{C_x}@T_x} \qquad \forall f, s. f \notin \Delta \\ T'_{s1} = funcArg(T_C@T_{STs1}, T_C@T_{this}, T_C@T'_{this}) \qquad \overline{T'_{s2} = funcArg(T_{s2}, T_x, T_{C_x}@T_{xST})}\end{array}}{\Gamma; \Delta, s_1 : T_C@T'_{STs1}, \overline{s_2 : T_{s2}} \vdash_s s_1.m\langle\overline{T_M}\rangle(\overline{s_2}) : T \dashv \Delta, s_1 : T'_{s1}, \overline{s_2 : T'_{s2}}}\ \text{T-INV.}$$

Private invocations differ from public invocations, because the current types of the fields must be checked against the transaction's preconditions and the field types must be updated after invocation,

$$\frac{\begin{array}{c}\Gamma \vdash bound(T_C@T'_{STs1}) = D\langle\overline{T}\rangle@T_{STs1} \\ specializeTrans_\Gamma(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle) = \overline{T_{C_f}@T_{fdecl} \gg T_{fST}\ x}\ T\ m(\overline{T_{C_x}@T_x \gg T_{xST}\ x})\ T_{this} \gg T'_{this}\ e \\ \Gamma \vdash T_{STs1} <:_* T_{this} \qquad \overline{\Gamma \vdash T_{s2} <: T_{C_x}@T_x} \qquad \overline{\Gamma \vdash T_f <: T_{C_f}@T_{fdecl}} \\ T'_{s1} = funcArg(C@T_{STs1}, C@T_{this}, C@T'_{this}) \qquad \overline{T'_{s2} = funcArg(T_{s2}, T_x, T_{C_x}@T_{xST})} \\ \overline{T'_f = funcArg(T_f, T_{C_f}@T_{fdecl}, T_{C_f}@T_{fST})}\end{array}}{\Gamma; \Delta, s_1 : T_C@T_{STs1}, \overline{s_2 : T_{s2}}, \overline{s.f : T_f} \vdash_{s_1} s_1.m\langle\overline{T_M}\rangle(\overline{s_2}) : T \dashv \Delta, s_1 : T'_{s1}, \overline{s_2 : T'_{s2}}, \overline{s.f : T'_f}}\ \text{T-PRIVINV.}$$

T-$\rightarrow_p$ allows changing the nominal state of this. Unlike transitions in FT, T-$\rightarrow_p$ does not permit arbitrary changes of class; it restricts the change to states within the object's current contract.

The T-$\rightarrow_p$ rule first checks that the current permission of this is compatible with p. Then, it checks that the values to be assigned to the new fields are compatible with the declared types of those fields. Finally, it ensures that all possible current fields of this that may reference assets do not currently own those assets. *StateFields* (AJ:1) looks up the fields defined in a particular state. The *fieldTypes* judgment (AJ:6) gives the current types of the input fields,

$$\frac{\begin{array}{c}\Gamma \vdash T_{ST} <:_* p \qquad p \in \{\text{Shared}, \text{Owned}\}] \qquad \Gamma; \Delta \vdash_s \overline{x} : \overline{T} \dashv \Delta' \\ \overline{\Gamma \vdash T <: stateFields(C\langle\overline{T_A}\rangle, S')} \qquad unionFields(C\langle\overline{T_A}\rangle, T_{ST}) = \overline{T_{fs}\ f_s} \\ fieldTypes_s(\Delta; \overline{T_{fs}\ f_s}) = \overline{T'_{fs}} \qquad \overline{\Gamma \vdash disposable(T'_{fs})}\end{array}}{\Gamma; \Delta, s : C\langle\overline{T_A}\rangle@T_{ST} \vdash_s s \rightarrow_p S'(\overline{x}) : \text{unit} \dashv \Delta', s : C\langle\overline{T_A}\rangle@S'}\ \text{T-}\rightarrow_p.$$

Checking static assertions of specific states is straightforward; we check to make sure that all possible states of the reference to be checked are among the permitted states,

$$\frac{\overline{S} \subseteq \overline{S'}}{\Gamma; \Delta, x : T_C@\overline{S} \vdash_s [x@\overline{S'}] : \text{unit} \dashv \Delta, x : T_C@\overline{S}}\ \text{T-ASSERTSTATES.}$$

When checking a static assertion of fixed permission, T-ASSERTPERMISSION checks for a precise match of permission,

$$\frac{T_{ST} \in \{\text{Owned}, \text{Unowned}, \text{Shared}\}}{\Gamma; \Delta, x : T_C@T_{ST} \vdash_s [x@T_{ST}] : \text{unit} \dashv \Delta, x : T_C@T_{ST}}\ \text{T-ASSERTPERMISSION.}$$

When asserting that a variable is in a state corresponding to a type variable, $bound_*$ is used to compute the most specific mode for the variable. T-ASSERTINVAR applies only if the computed bound is concrete (i.e., has no type variables), which is expressed by the *nonVar* judgment

(AJ:25). Otherwise, we require that the asserted mode be an exact match with the current type (T-assertInVarAlready),

$$\frac{\Gamma \vdash \mathit{bound}_*(p) = T_{ST} \qquad \mathit{nonVar}(T_{ST}) \qquad \Gamma;\Delta, x : T_C@T_{ST} \vdash_s [x@T_{ST}] : \mathsf{unit} \dashv \Delta, x : T_C@T_{ST}}{\Gamma;\Delta, x : T_C@T_{ST} \vdash_s [x@p] : \mathsf{unit} \dashv \Delta, x : T_C@T_{ST}}\ \textsc{T-assertInVar}$$

$$\frac{}{\Gamma;\Delta, x : T_C@p \vdash_s [x@p] : \mathsf{unit} \dashv \Delta, x : T_C@p}\ \textsc{T-assertInVarAlready}.$$

Dynamic state tests are typechecked according to the ownership of the variable to be checked. T-IsIn-StaticOwnership can be used when a variable is an owning reference but does not provide a particular state specification that the programmer wants. In contrast, IsIn-Dynamic applies when there is no ownership.

In T-IsIn-StaticOwnership, $T_C@T_{ST}$ is the type of the expression to be checked. The rule applies only when the reference to be checked is owned. T-IsIn-StaticOwnership ensures that the dynamic states to be checked are valid states according to the declaration of T_C. Then, e_1 is checked in a context that assumes that the test has passed; later, e_2 is checked in a context that assumes that the test failed. The resulting contexts of the two branches are merged together to construct an output context Δ_f. The *merge* judgment (AJ:33) merges two contexts. The *possibleStates* judgment (AJ:16) extracts all possible states given a type; the *states* judgment (AJ:8) extracts state definitions,

$$\frac{\overline{S} \subseteq \mathit{states}(T_C) \qquad \Gamma \vdash T_{ST} <:_* \mathsf{Owned} \qquad \Gamma;\Delta, x : T_C@\overline{S} \vdash_s e_1 : T_1 \dashv \Delta' \qquad \overline{S_x} = \mathit{possibleStates}_\Gamma(T_C@T_{ST}) \qquad \Gamma;\Delta, x : T_C@(\overline{S_x} \setminus \overline{S}) \vdash_s e_2 : T_1 \dashv \Delta'' \qquad \Delta_f = \mathit{merge}(\Delta', \Delta'')}{\Gamma;\Delta, x : T_C@T_{ST} \vdash_s \mathsf{if}\ x\ \mathsf{in}_{\mathit{owned}}\ \overline{S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta_f}\ \textsc{T-IsIn-StaticOwnership}.$$

In if x $\mathsf{in}_{\mathit{shared}}$ $\overline{S}$ then e_1 else e_2, e_1 is permitted to change the state of the object referenced by x, but it is not permitted to allow another reference to obtain permanent ownership of the object. While e_1 is evaluating, all state changes to the object referenced by x that occur via Shared aliases will cause program termination (due to a *dynamic lock*), so it is up to the programmer to ensure that this is impossible.

T-IsIn-Dynamic first checks that the states to be checked are well-defined. It checks e_1 in a context in which x has type $T_C@\overline{S}$, since the test passed if e_1 is evaluating. In contrast, the context for e_2 merely retains the Shared permission, since shared references cannot make any assumptions about state,

$$\frac{\overline{S} \subseteq \mathit{states}(T_C) \qquad \Gamma;\Delta, x : T_C@\overline{S} \vdash_s e_1 : T_1 \dashv \Delta', x : T_C@T'_{ST} \qquad \Gamma \vdash \mathit{bound}_*(T'_{ST}) \neq \mathsf{Unowned} \qquad \Gamma;\Delta, x : T_C@\mathsf{Shared} \vdash_s e_2 : T_1 \dashv \Delta'', x : T_C@\mathsf{Shared} \qquad \Delta_f = \mathit{merge}(\Delta', \Delta''), x : T_C@\mathsf{Shared}}{\Gamma;\Delta, x : T_C@\mathsf{Shared} \vdash_s \mathsf{if}\ x\ \mathsf{in}_{\mathit{shared}}\ \overline{S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta_f}\ \textsc{T-IsIn-Dynamic}.$$

If the test is against a permission variable, then T-IsIn-PermVar checks e_1 in a context that gives x the permission variable's permission, which will result in relying on the bound on p in Γ,

$$\frac{\Gamma;\Delta, x : T_C@p \vdash_s e_1 : T_1 \dashv \Delta' \qquad \Gamma;\Delta, x : T_C@T_{ST} \vdash_s e_2 : T_1 \dashv \Delta'' \qquad \Delta_f = \mathit{merge}(\Delta', \Delta'') \qquad \mathit{Perm} = \mathit{toPermission}(T_{ST})}{\Gamma;\Delta, x : T_C@T_{ST} \vdash_s \mathsf{if}\ x\ \mathsf{in}_{\mathit{Perm}}\ p\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta_f}\ \textsc{T-IsIn-PermVar}.$$

In T-IsIn-Perm-Then and T-IsIn-Perm-Else, the compiler knows which branch will be taken: Either T_{ST} satisfies the given condition or it does not. If T_{ST} is a variable, then it is treated as if it were owned (via *toPermission*),

$$\frac{\begin{array}{c}Perm \in \{\mathsf{Owned}, \mathsf{Unowned}, \mathsf{Shared}\} \qquad P = toPermission(T_{ST}) \\ \Gamma \vdash P <:_* Perm \qquad \Gamma; \Delta, x : T_C@T_{ST} \vdash_s e_1 : T_1 \dashv \Delta'\end{array}}{\Gamma; \Delta, x : T_C@T_{ST} \vdash_s \mathsf{if}\ x\ \mathsf{in}_P\ Perm\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta'}\ \textsc{T-IsIn-Perm-Then}$$

$$\frac{\begin{array}{c}Perm \in \{\mathsf{Owned}, \mathsf{Unowned}, \mathsf{Shared}\} \qquad P = toPermission(T_{ST}) \\ \Gamma \vdash P \not<:_* Perm \qquad \Gamma; \Delta, x : T_C@T_{ST} \vdash_s e_2 : T_1 \dashv \Delta'\end{array}}{\Gamma; \Delta, x : T_C@T_{ST} \vdash_s \mathsf{if}\ x\ \mathsf{in}_P\ Perm\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta'}\ \textsc{T-IsIn-Perm-Else}.$$

The case where a program tests to see if an unowned reference is in a particular state is included, because it can arise via substitution,

$$\frac{\Gamma; \Delta, x : T_C@\mathsf{Unowned} \vdash_s e_2 : T_1 \dashv \Delta'}{\Gamma; \Delta, x : T_C@\mathsf{Unowned} \vdash_s \mathsf{if}\ x\ \mathsf{in}_{\mathsf{Unowned}}\ \overline{S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \dashv \Delta'}\ \textsc{T-IsIn-Unowned}.$$

Disown discards ownership of its parameter. Existing ownership is split; in $T_C@T_{ST} \Rrightarrow T/T'$, T retains ownership and T' lacks it, so the output context uses T' as the new type of s'. Note that the split is not a function; one can see by inspection of the definition of split that T' is not owned, but may be either shared or unowned,

$$\frac{T_C@T_{ST} \Rrightarrow T/T' \qquad \Gamma \vdash T_{ST} <:_* \mathsf{Owned}}{\Gamma; \Delta, s' : T_C@T_{ST} \vdash_s \mathsf{disown}\ \ s' : \mathsf{unit} \dashv \Delta, s' : T'}\ \textsc{T-disown}.$$

pack updates Δ, removing all type overrides of fields of `this`. It requires that the existing overrides are consistent with the field declarations. There is no corresponding *unpack*; instead, field assignment and field reading can cause a future need to invoke pack. pack is defined in terms of $\approx$ (AJ:11), which ensures that either the two types are both owning or neither is owning. The *contractFields* judgment (AJ:5) extracts the fields that are available in all states of a contract,

$$\frac{s.f \notin dom(\Delta) \qquad contractFields(T) = \overline{T_{decl}\ f} \qquad \overline{\Gamma \vdash T_f <: T_{decl}} \qquad \overline{\Gamma \vdash T_f \approx T_{decl}}}{\Gamma; \Delta, s : T, \overline{s.f : T_f} \vdash_s \mathsf{pack} : \mathsf{unit} \dashv \Delta, s : T}\ \textsc{T-pack}.$$

$\boxed{M\ \textbf{ok in}\ C}$ Well-typed transaction

To check a public transaction, PublicTransactionOK first extracts the type parameters $\overline{T_G}$ of the enclosing contract C, the type variables in those parameters $\overline{T}$, and constructs a type bounds context Γ from T_G and T_M (the type parameters of the transaction). Then, the body of the transaction e is checked in a context that binds this and the parameters $\overline{x}$ to appropriate types. Those initial types come from the signature of the transaction; the final types in the output type context must match the specified types in the signature.

Note that all fields of this must end the transaction with types consistent with their declarations; otherwise, there would be occurrences of s.f in the output typing context after checking e. The body e may need to use pack to make this the case.

The *Var* judgment (AJ:27) extracts the type variables from type parameters. The *params* judgment (AJ:10) extracts the type parameters from declarations.

$$\frac{\begin{array}{c}params(C) = \overline{T_G} \qquad \overline{Var(T_G) = T} \qquad \Gamma = \overline{T_G}, \overline{T_M} \\ \Gamma; this : C\langle\overline{T}\rangle@T_{this},\ \overline{x : C_x@T_x} \vdash_{this} e : T \dashv this : C@T'_{this}, \overline{x : C_x@T'_x}\end{array}}{T\ m\langle\overline{T_M}\rangle(\overline{C_x@T_x \gg T'_x\ x})\ T_{this} \gg T'_{this}\ \{\mathsf{return}\ e\}\ \textbf{ok in}\ C}\ \textsc{PublicTransactionOK}.$$

The difference between public and private transactions is that private transactions may begin and end with fields inconsistent with their declarations. In both cases, inside e, it is possible to set fields of this so that they do not match their declared types. However, until the fields are updated so that their types match their declarations, additional public transactions cannot be invoked, ensuring that only private transactions are exposed to the inconsistent state.

There may be aliases to this. However, if the fields of this are inconsistent with their types, no public transactions can be invoked, so the inconsistency cannot be visible outside this transaction or any private transactions that it invokes. Furthermore, the state of this can only be changed if the permission on this allows that operation (see This-state-transition),

$$\frac{\begin{array}{c} params(C) = \overline{T_G} \qquad \overline{Var(T_G) = T} \qquad contractFields(C\langle\overline{T}\rangle) = \overline{T_f\ f} \\ \Delta = s : C\langle\overline{T}\rangle@T_{ST},\ \overline{s.f : contract(T_f).S_{f1}}, \overline{x : C_x@T_x} \\ \Delta' = s : C\langle\overline{T}\rangle@T'_{ST}, \overline{s.f : contract(T_f).S_{f2}}, \overline{x : C_x@T'_x} \\ \Gamma; \Delta \vdash_s e : T \dashv \Delta' \qquad \Gamma = \overline{T_G}, \overline{T_M} \end{array}}{\overline{S_{f1} >> S_{f2} f}\ T\ m\langle\overline{T_M}\rangle(C_x@T_x \gg T'_x x)\ T_{ST} \gg T'_{ST}\ \{\text{return } e\}\ \textbf{ok in } C}\ \textsc{PrivateTransactionOK}.$$

ST **ok** **Well-formed State**

All fields must have distinct names, and if any field is an asset, then the state must be labeled asset,

$$\frac{\forall i, j\ \ i \neq j \Rightarrow f_i \neq f_j \qquad \overline{\Gamma \vdash nonAsset(T)}}{\Gamma \vdash S\ \overline{T}\ \overline{f}\ \textbf{ok}} \qquad \frac{\forall i, j\ \ i \neq j \Rightarrow f_i \neq f_j}{\Gamma \vdash \text{asset } S\ \overline{T}\ \overline{f}\ \textbf{ok}}.$$

CL **ok** **Well-typed Contract**

Contracts must contain only methods and states that are well formed. Contracts must have at least one state, and every transaction specified in the interface that the contract claims to implement must have an implementation. Likewise, every state specified in the interface must be present in the contract. The *implementOk* judgments (AJ:28) define the relationships that must hold between interfaces, method signatures, and states. That is, all transactions and starts that are specified in the interface must be defined accordingly in the contract. Finally, the type parameters $\overline{T_G}$ of the contract must be well-formed (*genericsOk*, defined in AJ:30), and the type arguments given for the interface must be appropriate for the interface's specification (*subsOk*, defined in AJ:29).

The *isVar* judgment (AJ:26) identifies the types that are type variables. The *transactionName* function (AJ:7) extracts the name from a transaction declaration. The *stateNames* function (AJ:9) extracts names from declarations,

$$\frac{\begin{array}{c} \overline{M}\ \textbf{ok in } C \qquad \overline{T_G} \vdash \overline{ST}\ \textbf{ok} \qquad |\overline{ST}| > 0 \\ transactionNames(I) \subseteq transactionNames(C) \qquad stateNames(I) \subseteq stateNames(C) \\ \forall T \in \overline{T}, isVar(T) \implies T \in \overline{Var(T_G)} \\ \forall M \in \overline{M}, transactionName(M) \in transactionNames(I) \implies implementOk_{\overline{T_G}}(I\langle\overline{T}\rangle, M) \\ \forall S \in \overline{ST}, stateName(S) \in stateNames(I) \implies implementOk_{\overline{T_G}}(I\langle\overline{T}\rangle, S) \\ \overline{genericsOk_{\overline{T_G}}(T_G)} \qquad \overline{subsOk_{\overline{T_G}}(T, params(I))} \end{array}}{\text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\ \{\overline{ST}\ \overline{F}\ \overline{M}\}\ \textbf{ok}}.$$

IFACE **ok** **Well-typed Interface**
A well-typed interface must have well-formed type parameters (as defined by *genericsOk* in AJ:30),

$$\frac{\overline{genericsOk_{\overline{T_G}}(T_G)}}{\text{interface } I\langle\overline{T_G}\rangle\{\overline{ST}\ \overline{M_{SIG}}\}\ \textbf{ok}}.$$

PG **ok** **Well-typed Program**
A well-typed program consists of well-typed contracts, well-typed interfaces, and a well-typed expression,

$$\frac{\overline{CON}\ \textbf{ok} \qquad \overline{IFACE}\ \textbf{ok} \qquad \cdot;\cdot \vdash_s e : T \dashv \cdot}{\langle\overline{IFACE}, \overline{CON}, e\rangle\ \textbf{ok}}.$$

6.2 Subtyping

The subtyping relation depends on a sub-permission relation $<:_*$, defined below. The top-level subtyping relation mostly deals with type parameters, delegating the reasoning about permissions to $<:_*$. The definition uses the definition of substitution for types, which is defined in AJ:31.

$\Gamma \vdash \mathbf{T_1} <: \mathbf{T_2}$

$$\frac{}{\Gamma \vdash \text{unit} <: \text{unit}}\ <:\text{-Unit} \qquad \frac{\Gamma \vdash T_{ST} <:_* T'_{ST}}{\Gamma \vdash T_C@T_{ST} <: T_C@T'_{ST}}\ <:\text{-Matching-defs}$$

$$\frac{\Gamma \vdash T_{ST} <:_* T'_{ST}}{\Gamma \vdash D\langle\overline{T}\rangle@T_{ST} <: D\langle\overline{T}\rangle@T'_{ST}}\ <:\text{-Matching-decls}$$

$$\frac{\Gamma \vdash T_{ST} <:_* T'_{ST} \qquad def(C) = \text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T'}\rangle\{\ldots\}}{\Gamma \vdash C\langle\overline{T}\rangle@T_{ST} <: \sigma(\overline{T/T_G})(I\langle\overline{T'}\rangle@T'_{ST})}\ <:\text{-Implements-interface}$$

$$\frac{\Gamma \vdash T_{ST} <:_* T'_{ST} \qquad \Gamma \vdash bound(X@T_{ST}) = T_C@T'_{ST}}{\Gamma \vdash X@T_{ST} <: T_C@T'_{ST}}\ <:\text{-Bound}$$

$\Gamma \vdash T_{ST1} <:_* T_{ST2}$ **Subpermissions**
The subpermission judgment is ancillary to the subtyping judgment and specifies when an expression with one mode can be used where one with the same contract but a potentially different mode is expected,

$$\frac{}{\Gamma \vdash T_{ST} <:_* T_{ST}}\ <:_*\text{-Refl} \qquad \frac{\Gamma \vdash T_{ST_1} <:_* T_{ST_2} \qquad \Gamma \vdash T_{ST_2} <:_* T_{ST_3}}{\Gamma \vdash T_{ST_1} <: T_{ST_3}}\ <:_*\text{-Trans}$$

$$\frac{\Gamma \vdash bound_*(p) = T_{ST}}{\Gamma \vdash p <:_* T_{ST}}\ <:_*\text{-Var} \qquad \frac{\overline{S} \subseteq \overline{S'}}{\Gamma \vdash \overline{S} <:_* \overline{S'}}\ <:_*\text{-S-S'} \qquad \frac{}{\overline{S} <:_* \text{Owned}}\ <:_*\text{-S-O}$$

$$\frac{T_{ST2} \neq \overline{S}}{\Gamma \vdash \text{Owned} <:_* T_{ST2}}\ <:_*\text{-O-}^* \qquad \frac{}{\Gamma \vdash T_{ST} <:_* \text{Unowned}}\ <:_*\text{-U-U}$$

$\boxed{\Gamma \vdash T_{ST} \not<:_* T_{ST}}$

$$\frac{\Gamma \vdash T_{ST_2} <:_* T_{ST_1} \qquad T_{ST_2} \neq T_{ST_1}}{\Gamma \vdash T_{ST_1} \not<:_* T_{ST_2}}$$

6.3 Silica Dynamic Semantics

To express the dynamic semantics, we must first slightly extend the syntax. Objects, which reside in heaps μ, are referenced by object references o. We introduce a notion of *indirect references* l, which are used only in the formal semantics, not in the implementation, as a tool to prove soundness. Indirect references, introduced in FT [Garcia et al. 2014], allow the formal model to track permissions of individual aliases to shared objects. Intuitively, indirect references typically correspond with local variables that reference objects; we record the permission each indirect reference holds in a context ρ,

$$\begin{aligned}
&o \in \textsc{ObjectRefs}\\
&l \in \textsc{IndirectRefs}\\
C\langle\overline{T}\rangle@S(\overline{o}) &\in \textsc{Objects}\\
&\xi \in \textsc{PermissionVariables} \rightharpoonup \textsc{StateNames} \cup \{\mathsf{Owned}, \mathsf{Unowned}, \mathsf{Shared}\}\\
&\mu \in \textsc{ObjectRefs} \rightharpoonup \textsc{Objects}\\
&\rho \in \textsc{IndirectRefs} \rightharpoonup \textsc{Values}
\end{aligned}$$

e	::=	$\ldots \mid o$	
	$\mid$	$\boxed{e}_o$	(state-locking mutation detection container)
	$\mid$	$\boxed{e}^o$	(reentrancy detection container)
s	::=	$\ldots \mid l$	
v	::=	$() \mid o$	(values)
ϕ	::=	$\cdot \mid \phi, o$	(Objects that are state-locked)
ψ	::=	$\cdot \mid \psi, o$	(Objects that have transactions that are on the stack)
$\mathbb{E}$	::=	$\square$	
	$\mid$	let $x = \mathbb{E}$ in e	
	$\mid$	$\boxed{\mathbb{E}}_g$	
	$\mid$	$\boxed{\mathbb{E}}^g$	

We extend the previous definition of static contexts so that programs can remain well-typed as they execute:

$$\begin{array}{lll} b & \in & x \mid l \mid o \\ \Delta & ::= & \overline{b : T}. \end{array}$$

We extend the previous *T-lookup* rule to account for this extension:

$$\frac{T_1 \Rrightarrow T_2/T_3}{\Gamma; \Delta, b : T_1 \vdash_s b : T_2 \dashv \Delta, b : T_3}\ \textsc{T-lookup}.$$

The abstract machine maintains state $\langle \mu, \rho, \phi, \psi, \xi \rangle$. For concision, we abbreviate that tuple as Σ and refer to the components as Σ_μ, and so on. μ is used as an abbreviation for Σ_μ when there is only one Σ in scope. The syntax $[X/\mu]\,\Sigma$ denotes $\langle X, \rho, \phi, \psi, \xi \rangle$; μ denotes Σ_μ if it occurs in X. In addition, we use $\Sigma[l \mapsto o]$ to denote an extension of Σ_ρ.

The dynamic semantics are similar to the dynamic semantics of FT. However, in addition to heap μ and environment ρ, we keep a state-locking environment ϕ, which is a set of references to objects that are state-locked. ψ is used for dynamic reentrancy detection; ξ keeps track of which states and permissions are feasible for each permission variable.

$\boxed{\Sigma, e \to \Sigma', e'}$

$$\frac{}{\Sigma, l \to \Sigma, \rho(l)}\ \text{E-LOOKUP} \qquad \frac{o = \rho(l')}{\Sigma, l := l' \to [\rho[l \mapsto o]/\rho]\,\Sigma, ()}\ \text{E-ASSIGN}$$

$$\frac{l \notin dom(\rho)}{\Sigma, \text{let } x : T = v \text{ in } e \to [\rho[l \mapsto v]/\rho]\,\Sigma, [l/x]e}\ \text{E-LET}$$

$$\frac{\Sigma, e_1 \to \Sigma', e_1'}{\Sigma, \text{let } x : T = e_1 \text{ in } e_2 \to \Sigma', \text{let } x : T = e_1' \text{ in } e_2}\ \text{E-LETCONGR}$$

$$\frac{o \notin dom(\mu) \qquad def(C) = \text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\{\ldots\}}{\Sigma, \text{new } C\langle\overline{T}\rangle@S(\bar{l}) \to [\mu[o \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l)})]/\mu]\,\Sigma, o}\ \text{E-NEW}$$

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\bar{s})}{\Sigma, l.f_i \to \Sigma, s_i}\ \text{E-FIELD}$$

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\bar{l}) \qquad fields(C@S) = \overline{T\ f}}{\Sigma, l.f_i := l' \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(o_1, o_2, \ldots, o_{i-1}, \rho(l'), o_{i+1}, \ldots, o_{|l|})]/\mu]\,\Sigma, ()}\ \text{E-FIELDUPDATE}$$

The two invocation rules are complex. Reentrancy is checked dynamically at object granularity. Object-level reentrancy aborts the current top-level transaction. However, as a special exception, private transactions are not protected from reentrancy (otherwise they would be useless). Reentrancy is checked via the ψ context, which is a set of all objects that have transaction invocations on the stack.

First, we look up the receiver in the heap to find its dynamic state. In invocations of public methods, we also must check (by looking in ψ) that there is not already an invocation on the receiver in progress. Then, we make fresh indirect references l_1' and $\overline{l_2'}$, which will be used to pass ownership to the transaction; residual ownership will remain in the original indirect references l_1 and $\overline{l_2}$. Then, since e may use type parameters according to the declarations of C and $tdef(C, m)$, we need to update ξ so that the variables are bound according to the invocation by resolving any type variables to concrete permissions or states (via *lookup*). Then, we proceed by substitution in an environment that tracks in ψ an in-progress invocation on the object indirectly referenced via l_1, which is directly referenced by $\rho(l_1)$. This object reference must be removed from ψ when evaluation of the transaction body is complete. To arrange this, the rule steps to an expression in a box. Afterward, evaluation will proceed inside the box until the contents of the box reaches a value, at which point the invocation returns, the value is unboxed, and the reference is removed from ψ.

We define $lookup_\xi(T_{ST})$ so that it looks up T_{ST} in ξ if T_{ST} is a variable and otherwise simply maps to T_{ST}. This definition ensures that each permission variable maps to a concrete permission

or state, rather than a permission variable, eliminating the need for recursive lookups.

$$\frac{\begin{array}{c}\mu(\rho(l_1)) = C\langle\overline{T}\rangle@S(\ldots) \qquad \rho(l_1) \notin \psi \\ tdef(C,m) = T\ m\langle\overline{T_M}\rangle(\overline{T_{C_x}@T_x \gg T_{xST}\ x})T_{this} \gg T'_{this}\ \{\text{return } e\} \\ l'_1 \notin dom(\rho) \qquad \overline{l'_2 \notin dom(\rho)} \qquad params(C) = \overline{T_D} \\ \xi' = \xi, \overline{PermVar(T_D) \mapsto lookup_\xi(Perm(T))}, \overline{PermVar(T_M) \mapsto lookup_\xi(Perm(M))} \\ \rho' = \rho, l'_1 \mapsto \rho(l_1), \overline{l'_2 \mapsto \rho(l_2)} \\ \Sigma' = \langle\mu, \rho', \phi, (\psi, \rho(l_1)), \xi'\rangle\end{array}}{\Sigma, l_1.m\langle\overline{M}\rangle(\overline{l_2}) \to \Sigma', \boxed{\overline{[l'_2/x]}[l'_1/\text{this}]e}^{\rho(l_1)}}\ \text{E-INV}$$

$$\frac{\begin{array}{c}\mu(\rho(l_1)) = C\langle\overline{T}\rangle@S(\ldots) \\ tdef(C,m) = \overline{T_{C_f}@T_{fdecl} \gg T_{fST}}\ T\ m\langle\overline{T_M}\rangle(\overline{T_{C_x}@T_x \gg T_{xST}\ x})\ T_{this} \gg T'_{this}\ \{\text{return } e\} \\ l'_1 \notin dom(\rho) \qquad \overline{l'_2 \notin dom(\rho)} \qquad params(C) = \overline{T_D} \\ \xi' = \xi, \overline{PermVar(T_D) \mapsto lookup_\xi(Perm(T))}, \overline{PermVar(T_M) \mapsto lookup_\xi(Perm(M))} \\ \rho' = \rho, l'_1 \mapsto \rho(l_1), \overline{l'_2 \mapsto \rho(l_2)} \\ \Sigma' = \langle\mu, \rho', \phi, \psi, \xi'\rangle\end{array}}{\Sigma, l_1.m\langle\overline{M}\rangle(\overline{l_2}) \to \Sigma', \overline{[l'_2/x]}[l'_1/\text{this}]e}\ \text{E-PRIVINV}$$

The two rules above use $lookup_\xi$:

$$\boxed{lookup_\xi(T_{ST}) = T_{ST}}$$

$$\frac{}{lookup_\xi(p) = \xi(p)} \qquad \frac{nonVar(T_{ST})}{lookup_\xi(T_{ST}) = T_{ST}}$$

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots)}{\Sigma, l \to_{owned} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()}\ \text{E-}\to_{owned}.$$

In E-$\to_{shared}$, a shared object can transition state if it is not statelocked ($\rho(l) \notin \phi$) or the transition does not actually change which state the object is in,

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots) \qquad \rho(l) \notin \phi \vee S = S'}{\Sigma, l \to_{shared} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()}\ \text{E-}\to_{shared}$$

$$\frac{}{\Sigma, [s@T_{ST}] \to \Sigma, ()}\ \text{E-ASSERT}.$$

In the scope of an `if in` block, we must ensure that other aliases cannot be used to violate the state assumptions of the block. We only check for state modification, not for general field writes, since the typestate mechanism is restricted to nominal states rather than pertaining to all properties of objects. To do this, we track references to objects that are state-locked in ϕ. These references are dynamically state-locked: Modifications to the referenced objects' state via *other*

references will result in an expression getting stuck.

$$\frac{\xi(p) = T_{ST}}{\Sigma, \text{if } l \text{ is in}_\text{P} \ p \text{ then } e_1 \text{ else } e_2 \to \Sigma, \text{if } l \text{ is in}_\text{P} \ T_{ST} \text{ then } e_1 \text{ else } e_2} \text{ E-IsIn-PermVar}$$

$$\frac{\text{Perm} \in \{\text{Owned}, \text{Unowned}, \text{Shared}\} \qquad \cdot \vdash P <:_* \text{Perm}}{\Sigma, \text{if } l \text{ is in}_\text{P} \text{ Perm then } e_1 \text{ else } e_2 \to \Sigma, e_1} \text{ E-IsIn-Perm-Then}$$

$$\frac{\text{Perm} \in \{\text{Owned}, \text{Unowned}, \text{Shared}\} \qquad \cdot \vdash \text{Perm} \not<:_* P}{\Sigma, \text{if } l \text{ is in}_\text{P} \text{ Perm then } e_1 \text{ else } e_2 \to \Sigma, e_2} \text{ E-IsIn-Perm-Else}$$

$$\frac{}{\Sigma, \text{if } l \text{ is in}_\text{Unowned} \ \overline{S} \text{ then } e_1 \text{ else } e_2 \to \Sigma, e_2} \text{ E-IsIn-Unowned}$$

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots) \qquad S' \in \overline{S}}{\Sigma, \text{if } l \text{ is in}_\text{owned} \ \overline{S} \text{ then } e_1 \text{ else } e_2 \to \Sigma, e_1} \text{ E-IsIn-Owned-Then}$$

E-IsIn-Shared-Then checks $\rho(l) \notin \phi$, because the static semantics that correspond generate a temporary owning reference. If it did not check, or it allowed nested checks, then that would generate multiple distinct temporary owning references

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots) \qquad \rho(l) \notin \phi}{\Sigma, \text{if } l \text{ is in}_\text{shared} \ \overline{S} \text{ then } e_1 \text{ else } e_2 \to [\phi, \rho(l)/\phi] \ \Sigma, \boxed{e_1}_{\rho(l)}} \text{ E-IsIn-Shared-Then}$$

$$\frac{\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots) \qquad S' \notin \overline{S}}{\Sigma, \text{if } l \text{ is in}_p \ \overline{S} \text{ then } e_1 \text{ else } e_2 \to \Sigma, e_2} \text{ E-IsIn-Else} \qquad \frac{}{\Sigma, \text{disown } s \to \Sigma, ()} \text{ E-disown}$$

$$\frac{}{\Sigma, \text{pack} \to \Sigma, ()} \text{ E-pack}$$

$\boxed{e}_o$ and $\boxed{e}^o$ permit the boxed expression to first evaluate to a value, and then afterward remove the corresponding object reference from the appropriate context,

$$\frac{\Gamma; \Delta \vdash_s e : T \dashv \Delta'}{\Gamma; \Delta \vdash_s \boxed{e}_o : T \dashv \Delta'} \text{ T-state-mutation-detection} \qquad \frac{\Gamma; \Delta \vdash_s e : T \dashv \Delta'}{\Gamma; \Delta \vdash_s \boxed{e}^o : T \dashv \Delta'} \text{ T-reentrancy-detection}$$

$$\frac{}{\Sigma, \boxed{v}_o \to [(\phi \setminus o)/\phi] \ \Sigma, v} \text{ E-box-}\phi \qquad \frac{}{\Sigma, \boxed{v}^o \to [(\psi \setminus o)/\psi] \ \Sigma, v} \text{ E-box-}\psi$$

$$\frac{\Sigma, e \to \Sigma', e'}{\Sigma, \boxed{e}_o \to \Sigma', \boxed{e'}_o} \text{ E-box-}\phi\text{-congr} \qquad \frac{\Sigma, e \to \Sigma', e'}{\Sigma, \boxed{e}^o \to \Sigma', \boxed{e'}^o} \text{ E-box-}\psi\text{-congr.}$$

6.4 Silica Soundness and Asset Retention

In this section, we outline the proof of type soundness. We also state the *asset retention* theorem, which formally states the property that owned references to assets can only be dropped with the disown operation. Full proofs can be found in Appendix C. We focused on a paper-based proof

rather than a mechanized proof due to the high cost of mechanization. We derive additional confidence in the soundness from the existing soundness proofs of related systems, including FT [Garcia et al. 2014].

Global consistency defines consistency among static and runtime environments. It requires that every indirect reference to an object in ρ maps to a legitimate indirect reference in μ and that ρ maps indirect references to appropriately typed values. It also requires that every type in the static context correspond with an indirect reference in the indirect reference context. The permission variables must be available for lookup in ξ and map to concrete permissions or states. Finally, every object in the heap must have only compatible aliases, as expressed by *reference consistency*.

We will also need typing for ():

$$\frac{}{\Gamma; \Delta \vdash_s () : \mathsf{unit} \dashv \Delta}\ \text{T-()}.$$

$\boxed{\Gamma, \Sigma, \Delta\ \mathbf{ok}}$ Global Consistency

$$\frac{\begin{array}{c} range(\rho) \subset dom(\mu) \cup \{()\} \\ dom(\Delta) \subset dom(\rho) \cup dom(\mu) \\ \{l \mid (l : \mathsf{unit}) \in \Delta\} \subset \{l \mid \rho(l) = ()\} \\ \{l \mid (l : boolean) \in \Delta\} \subset (\{l \mid \rho(l) \in \{true, false\}\} \\ \{l \mid (l : T) \in \Delta\} \subset \{l \mid \rho(l) = o\} \\ PermVar(\Gamma) \subset \{p \mid \xi(p) = T_{ST}\} \\ \forall s : T_C@T_{ST} \in \Delta, \exists C, \overline{T} \text{ s.t. } T_C = C\langle\overline{T}\rangle \\ \Sigma, \Delta \vdash dom(\mu)\ \mathbf{ok} \end{array}}{\Gamma, \Sigma, \Delta\ \mathbf{ok}}.$$

Reference consistency expresses the requirement that all aliases to a given object must be compatible with each other and consistent with the actual type of the object in the heap. It also requires that objects in the heap have the right number of fields. The fact that the fields must reference objects of appropriate type is implied by the requirement that all references must reference objects of types consistent with the reference types.

$\boxed{\Sigma, \Delta \vdash o\ \mathbf{ok}}$ Reference Consistency

$$\frac{\begin{array}{c} \mu(o) = C\langle\overline{T}\rangle@S(\overline{o'}) \qquad |\overline{o'}| = |stateFields(C, S)| \\ refTypes(\Sigma, \Delta, o) = \overline{D} \qquad \cdot \vdash C\langle\overline{T}\rangle <: \overline{D} \\ \forall T_1, T_2 \in \overline{D}, T_1 \leftrightarrow T_2 \text{ or } StateLockCompatible(T_1, T_2) \end{array}}{\Sigma, \Delta \vdash o\ \mathbf{ok}},$$

where

$$StateLockCompatible(T_1, T_2) \triangleq o \in \Sigma_\phi \wedge ((i \neq j) \implies owned(T_i) \wedge T_j = C\langle\overline{T}\rangle@Shared).$$

StateLockCompatible is defined to allow the original Shared alias (via which the state was checked) to co-exist with the state-specifying reference. This would not normally be permitted, but is safe while $o \in \Sigma_\phi$, because the shared alias cannot be used to mutate typestate while that is the case.

The relation $\leftrightarrow$ defines compatibility between pairs of aliases:

$\boxed{T_1 \leftrightarrow T_2}$ Alias Compatibility

$$\frac{T_2 \leftrightarrow T_1}{T_1 \leftrightarrow T_2}\ \textsc{SymCompat} \qquad \frac{C\langle\overline{T}\rangle@T_{ST} \leftrightarrow C\langle\overline{T}\rangle@T'_{ST}}{C\langle\overline{T}\rangle@T_{ST} \leftrightarrow I\langle\overline{T}\rangle@T_{ST}}\ \textsc{SubtypeCompat}$$

$$\frac{C\langle\overline{T}\rangle@T_{ST} \leftrightarrow C\langle\overline{T}\rangle@T'_{ST}}{C\langle\overline{T}\rangle@T_{ST} \leftrightarrow C\langle\overline{T'}\rangle@T'_{ST}}\ \textsc{ParamCompat} \qquad \frac{}{T_C@Unowned \leftrightarrow T_C@Unowned}\ \textsc{UUCompat}$$

$$\frac{}{T_C@Unowned \leftrightarrow T_C@Shared}\ \textsc{USCompat} \qquad \frac{}{T_C@Unowned \leftrightarrow T_C@Owned}\ \textsc{UOCompat}$$

$$\frac{}{T_C@Unowned \leftrightarrow T_C@\overline{S}}\ \textsc{UStatesCompat} \qquad \frac{}{T_C@\mathsf{Shared} \leftrightarrow T_C@\mathsf{Shared}}\ \textsc{SCompat}$$

$refTypes$ computes the set of types of referencing aliases to a given object in a given static and dynamic context. References may be from fields of objects in the heap; from indirect references; and from variables in the static context. Fields of objects in the heap include both fields whose types are specified in their declarations and fields whose types are overridden temporarily in the static context Δ,

$$\begin{aligned}
refTypes(\Sigma, \Delta, o) &= refFieldTypes(\mu, o) +\!\!+ envTypes(\Sigma, o) +\!\!+ ctxTypes(\Delta, o)\\
refFieldTypes(\mu, o) &= \mathop{+\!\!+}_{o' \in dom(\mu)} [T_i \mid \mu(o') = C\langle\overline{T}\rangle@S(\overline{o}), ft(\Delta, C\langle\overline{T}\rangle, S) = \overline{Tf} \text{ and } o \in \overline{o}]\\
ft(\Delta, C, S) &= [Tf \mid s.f : T \in \Delta] \cup (allFields(C, S) \setminus [Tf \mid s.f \in dom(\Delta)])\\
envTypes(\Sigma, \Delta, o) &= \mathop{+\!\!+}_{l \in dom(\rho)} [T \mid \Sigma_\rho(l) = o \text{ and } (l : T) \in \Delta]\\
ctxTypes(\Delta, o) &= [T \mid o : T \in \Delta].
\end{aligned}$$

Definition 6.1 ($<^l$). A context Δ is l-stronger than a context Δ' with respect to Γ and Σ (denoted $\Delta <^l_{\Gamma,\Sigma} \Delta'$) if and only if for all $l' : T' \in \Delta'$, there is some T and l such that $\Gamma \vdash T <: T'$, $l : T \in \Delta$, $T \approx T'$, and $\Sigma'_\rho(l) = \Sigma'_\rho(l')$.

Note that this differs from the definition of $<^l$ given in FT. Here, the indirect reference in the two contexts need not match. This weakening is necessary, because permissions are split in invocations. After an invocation, the new expression typechecks in a context that may retain some of the permissions from the original reference (whereas the remaining permissions were transferred to the invocation, i.e., retained in a *different* indirect reference). This means that although the permissions are still at least as strong in the new context, the strongest permission may be held by a different indirect reference than in the original.

Corollary 6.1 ($<^l$-reflexivity). *For all* Δ, Γ, Σ, $\Delta <^l_{\Gamma,\Sigma'} \Delta$.

Proof. Trivial application of the definition because $<:$ is reflexive. □

Lemma 6.1 (Canonical forms). *If* $\Gamma; \Delta \vdash_s v : T \dashv \Delta'$, *then:*

(1) If $T = C\langle\overline{T'}\rangle.\overline{S}$, *then* $v = C\langle\overline{T'}\rangle@S(\overline{s})$.
(2) If $T = unit$, *then* $v = ()$.

Proof. By inspection of the typing rules. □

Lemma 6.2 (Memory consistency). *If* Γ, Σ, Δ ***ok****, then:*

(1) If $l : C\langle\overline{T'}\rangle@S \in \Delta$*, then* $\exists o.\rho(l) = o$ *and* $\mu(o) = C\langle\overline{T'}\rangle@S(\overline{s})$.
(2) If $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$*, and* l *is a free variable of* e*, then* $l \in dom(\rho)$.

Proof. The proof is shown in Appendix C.1. □

Theorem 6.2 (Progress). *If* e *is a closed expression and* $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$*, then at least one of the following holds:*

(1) e *is a value*
(2) For any environment Σ *such that* Γ, Σ, Δ *ok,* $\Sigma, e \to \Sigma', e'$ *for some environment* Σ'
(3) e *is stuck at a bad state transition — that is,* $e = \mathbb{E}[l \nearrow_{Shared} S(\overline{s})]$ *where* $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S'(\ldots)$, $S \neq S'$, $\rho(l) \in \phi$, *and* $\Gamma; \Delta \vdash_s l : C\langle\overline{T'}\rangle@Shared \dashv \Delta'$.
(4) e *is stuck at a reentrant invocation – that is,* $e = \mathbb{E}[l.m(\overline{s})]$ *where* $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\ldots)$, $\rho(l) \in \psi$.
(5) e *is stuck in a nested dynamic state check – that is,* $e = \mathbb{E}[if\ s\ in_{shared}\ T_{ST}\ then\ e_1\ else\ e_2]$ *where* $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$ *and* $\rho(l) \in \phi$.

Proof. The proof, which proceeds by induction on the typing derivation, can be found in Appendix C. □

Theorem 6.3 (Preservation). *If* e *is a closed expression,* $\Gamma; \Delta \vdash_s e : T \dashv \Delta''$, Γ, Σ, Δ ***ok****, and* $\Sigma, e \to \Sigma', e'$ *then for some* Δ', Γ'; $\Delta' \vdash_s e' : T' \dashv \Delta'''$, $\Gamma', \Sigma', \Delta'$ ***ok****, and* $\Delta''' <^{l}_{\Gamma,\Sigma'} \Delta''$.

Proof. The proof, which proceeds by induction on the dynamic semantics, can be found in Appendix C. □

Informally, *asset retention* is the property that if a well-typed expression e takes a step in an appropriate dynamic context, then owning references to assets are only dropped if e is a disown operation.

Theorem 6.4 (Asset retention). *Suppose:*

(1) Γ, Σ, Δ ***ok***
(2) $o \in dom(\mu)$
(3) $refTypes(\Sigma, \Delta, o) = \overline{D}$
(4) $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$
(5) e *is closed*
(6) $\Sigma, e \to \Sigma', e'$
(7) $refTypes(\Sigma', \Delta', o) = \overline{D'}$
(8) $\exists T' \in \overline{D}$ *such that* $\Gamma \vdash nonDisposable\,(T')$
(9) $\forall T' \in \overline{D'} : \Gamma \vdash disposable\,(T')$

Then in the context of a well-typed program, either $\Gamma \vdash nonDisposable\,(T)$ *or* $e = \mathbb{E}[disown\ \ s]$, *where* $\rho(s) = o$.

Proof. The proof, which proceeds by induction on the typing derivation, can be found in Appendix C. □

7 OBSIDIAN LANGUAGE DEFINITION

Unlike Silica, which is an expression language to facilitate writing proofs, we designed Obsidian as a statement language to reflect the expectations of object-oriented programmers.

Table 3. Differences between Obsidian and Silica

Difference	Translation
Obsidian supports sequences of statements, not just expressions.	Sequences of statements are translated to a nested let-bind expression.
Constructors can have arbitrary behavior, not just initializing fields.	Constructors are translated to transactions that invoke `new` and return the result. Every use of `new` in Obsidian is translated to a call to that transaction.
A-normal form is not required.	The translator let-binds nested translated expressions to fresh variables and uses those variables in the body as required.
State transitions are not labeled with the type of `this`.	The translator locally infers types of `this` for state transitions [Pierce and Turner 2000].
State transitions specify new values for an arbitrary collection of fields.	The translator ensures that all new fields are initialized.
State tests are not labeled with the type of the expression being tested.	The translator locally infers types [Pierce and Turner 2000].
There is no pack in Obsidian.	pack is inserted automatically by the translator before public invocations and at the ends of transaction bodies.
Obsidian uses `S::f` to denote a field in a future state of `this`.	The translator defines local variables in a hidden namespace and inserts them into transitions.
There is no `switch` in Silica.	The translator translates each use of `switch` to a tree of `if in` expressions.
Local variable declarations in Obsidian specify contract types but may omit intializations for the new variables. Silica uses `let`.	A translation context tracks the declared contract types of variables and checks that assignments are consistent with the declarations. Translating local variable declarations that lack initialization mutates the context but does not emit any Silica code.

The high-level Obsidian language differs from the core Silica language in various ways discussed in this section. We define the semantics of Obsidian by translation to Silica, so Table 3 shows the differences and how the translator handles them.

7.1 Obsidian Syntax

In addition to the syntactic features described in the grammar below, the implementation also supports the following features:

- FFI contracts, which define interfaces that are implemented in trusted Java code rather than in Obsidian code (used for exposing runtime support in Fabric and JDK classes).
- Import statements, for using code defined in other files.

Also, note that in some cases the syntax is more permissive than strictly needed so that the compiler can conveniently give good error messages. For example, $e_1 = e_2$ is only permitted when e_1 is a variable, `S::f`, or `this.f`.

Relative to Silica, Obsidian adds constructors and statements:

CTOR	::=	C@P ($\overline{T \gg T_{ST}\ x}$) { $\overline{STMT}$ }	
STMT	::=	e	
	|	$e = e$	(variable assignment and field update)
	|	$\rightarrow S(\overline{f = e})$	(State transition)
	|	$C\ x = e$	(variable declaration with init)
	|	$C\ x$	(variable declaration)
	|	[e @ T_{ST}]	(static assert)
	|	if (e) {$\overline{STMT}$} else {$\overline{STMT}$}	
	|	if s in T_{ST} {$\overline{STMT}$} else {$\overline{STMT}$}	(dynamic state test)
	|	return [e]	
	|	revert e	(revert state, providing an error message)
	|	switch e{case $S\{\overline{STMT}\}$}	(case analysis on dynamic state)

To expressions, Obsidian adds assignments $e_1 = e_2$ as well as S::f, which represents a field `f` in a future state `S`. Obsidian also uses `new` to invoke constructors, rather than initializing fields directly.

e	::=	$e = e$	
	|	new $C(\overline{e})$	
	|	S::f	(state field initialization expression)

Although not shown here, the Obsidian implementation also supports primitive types `int`, `string`, and `bool`, constants of those types, and some primitive operators for expressions of those types, such as `+` and `-`.

We briefly outline a translation from Obsidian to Silica in Appendix B. The statement-oriented reformulation generally only results in trivial extensions to the typing rules. The most significant extension is for `if`, which, like dynamic state checks `if in`, merges the states from the two branches to construct a result context. When `return` occurs, the type checker conducts the same checks that run at the ends of transactions to ensure that no assets are lost. In addition, branches that are guaranteed to return or revert do not affect the typing context after the `if` statement. This optimization improves precision, since the code after the `if` only executes if the non-exiting branch is the one that executed, and the two branches might change the typestate of a variable divergently. Other than these small changes, we rely on Silica to show soundness of Obsidian.

Obsidian relaxes Silica's requirement that every contract define at least one state. Obsidian contracts that do not define states behave as if they were always in a particular state with a user-inaccessible state name.

In Obsidian, private transactions need not specify types for all fields. Instead, fields whose types are unspecified have types according to their declarations.

8 CASE STUDY EVALUATION

We wanted to ensure that Obsidian can be used to specify typical smart contracts in a concise and reasonable way. Therefore, we undertook two case studies to assess the extent to which Obsidian is suitable for implementing appropriate smart contracts.

Obsidian's type system has significant implications for the design and implementation of software relative to a traditional object-oriented language. We were interested in evaluating several research questions using the case studies:

(RQ1) Does the aliasing structure in real blockchain applications allow use of ownership (and therefore typestate)? If so, then what are the implications on architecture? Or,

alternatively, do so many objects need to be Shared that the main benefit of typestate is that it helps ensure that programmers insert dynamic tests when required?
(RQ2) To what extent does the use of typestate reduce the need for explicit state checks and assertions, which would otherwise be necessary?
(RQ3) Can realistic systems be built with Obsidian?
(RQ4) To what extent do realistic systems have constructs that are naturally expressed as states and assets?

To address the research questions above, we were interested in implementing blockchain applications in Obsidian. To obtain realistic results, we looked for domains in which:

- Use of a blockchain platform for the application would provide significant advantages over a traditional, centralized platform.
- We could engage with a real client to ensure that the requirements were driven by real needs, not by convenience of the developer or by the appropriateness of language features.
- The application seemed likely to be representative in structure of a large class of blockchain applications.

8.1 Case Study 1: Parametric Insurance

8.1.1 Motivation. In *parametric insurance*, a buyer purchases a claim, specifying a *parameter* that governs when the policy will pay out. The parameter is chosen so that whether conditions satisfy the parameter can be determined objectively. For example, a farmer might buy drought insurance as parametric insurance, specifying that if the soil moisture index (a property derived from weather conditions) in a particular location drops below m in a particular time window, the policy should pay out. The insurance is then priced according to the risk of the specified event. In contrast, traditional insurance would require that the farmer summon a claims adjuster, who could exercise subjective judgment regarding the extent of the crop damage. Parametric insurance is particularly compelling in places where the potential policyholders do not trust potential insurers, who may send dishonest or unfair adjusters. In that context, potential policyholders may also be concerned with the stability and trustworthiness of the insurer: what if the insurer pockets the insurance premium and goes bankrupt, or otherwise refuses to pay out legitimate claims?

To build a trustworthy insurance market for farmers in parts of the world without trust between farmers and insurers, the World Bank became interested in deploying an insurance marketplace on a blockchain platform. We partnered with the World Bank to use this application as a case study for Obsidian. We used the case study both to *evaluate* Obsidian as well as to *improve* Obsidian, and we describe below results in both categories.

The case study was conducted primarily by an undergraduate who was not involved in the language design, with assistance and later extensions by the language designers. The choice to have an undergraduate do the case study was motivated by the desire to learn about what aspects of the language were easy or difficult to master. It was also motivated by the desire to reduce bias; a language designer studying their own language might be less likely to observe interesting and important problems with the language.

We met regularly with members of the World Bank team to ensure that our implementation would be consistent with their requirements. We began by eliciting requirements, structured according to their expectations of the workflow for participants.

8.1.2 Requirements. The main users of the insurance system are *farmers*, *insurers*, and *banks*. Banks are necessary to mediate financial relationships among the parties. We assume that farmers

have local accounts with their banks and that the banks can transfer money to the insurers through the existing financial network. Basic assumptions of trust drove the design:

- Farmers trust their banks, with whom they already do business, but do not trust insurers, who may attempt to pocket their premiums and disappear without paying out policies.
- Insurers do not trust farmers to accurately report on the weather; they require a trusted weather service to do that. They do trust the implementation of the smart contracts to pay out claims when appropriate and to otherwise refund payout funds to the insurers at policy expiration.
- There exists a mutually trusted weather service, which can provide signed evidence of weather events.

8.1.3 Design. Because blockchains typically require all operations to be deterministic and all transactions to be invoked externally, we derived the following design:

- Farmers are responsible for requesting claims and providing acceptable proof of a relevant weather event to receive a payout.
- Insurers are responsible for requesting refunds when policies expire.
- A trusted, off-blockchain weather service is available that can, on request, provide signed weather data relevant to a particular query.

An alternative approach would involve the weather service handling weather subscriptions. The blockchain insurance service would emit events indicating that it subscribed to particular weather data, and the weather service would invoke appropriate blockchain transactions when relevant conditions occurred. However, this design is more complex and requires trusting the weather service to push requests in a timely manner. Our design is simpler but requires that policyholders invoke the claim transactions, passing appropriate signed weather records.

Our design of the application allows farmers to start the exchange by requesting bids from insurers. Then, to offer a bid, insurers are required to specify a premium and put the potential payout in escrow; this ensures that even if the insurer goes bankrupt later, the policy can pay out if appropriate. If the farmer chooses to purchase a policy, then the farmer submits the appropriate payment.

Later, if a weather event occurs that would justify filing a claim, then a farmer requests a signed weather report from the weather service. The farmer submits a claim transaction to the insurance service, which sends the virtual currency to the farmer. The farmer could then present the virtual currency to their real-world bank to enact a deposit.

8.1.4 Results. The implementation consists of 545 non-comment, non-whitespace lines of Obsidian code. For simplicity, the implementation is limited to one insurer, who can make one bid on a policy request. An overview of the invocations that are sent and results that are received in a typical successful bid and claim scenario is shown in Figure 9. All of the objects reside in the blockchain except as noted. The full code for this case study is available online.[1]

We made several observations about Obsidian. In some cases, we were able to leverage our observations to improve the language. In others, we learned lessons about the implications of the type system on application design and architecture.

First, in the version of the language that existed when the case study started, Obsidian included an *explicit ownership transfer operator* `<-`. In that version of the language, passing an owned reference as an argument would only transfer ownership to the callee if the argument was

[1] https://github.com/mcoblenz/Obsidian/tree/master/resources/case_studies/Insurance.

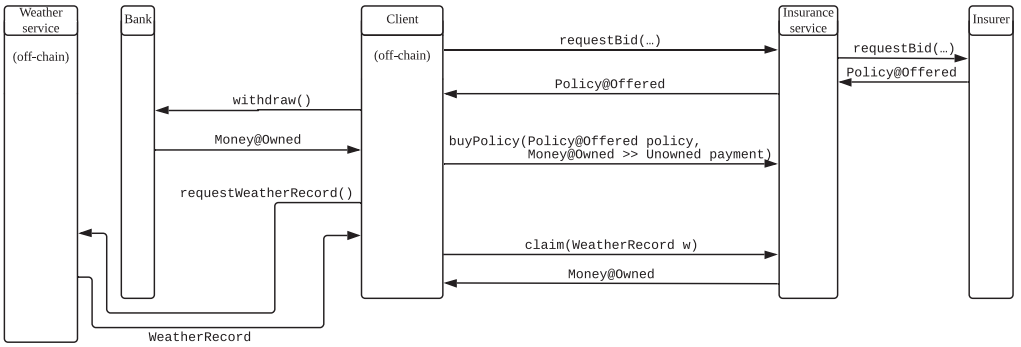

Fig. 9. Invocations sent and results returned in a typical successful bid/claim scenario.

decorated with `<-`. For example, `deposit(<-m)` would transfer ownership of the reference `m` to the `deposit` transaction, but `deposit(m)` would be a type error, because `deposit` requires an Owned reference. While redundant with type information, we had included the `<-` operator, because we thought it would reduce confusion, but we noticed while using the language (both in the case study and in smaller examples) that its presence was onerous. We removed it, which was a noticeable simplification.

Second, in that version of the language, `asset` could only be a property of contracts. We noticed in the insurance case study that it is more appropriate to think of `asset` as a property of states, since some states own assets and some do not. In the case study, an instance of the `PolicyRecord` contract holds the insurer's money (acting as an escrow) while a policy is active, but after the policy is expired or paid, the contract no longer holds money (and therefore no longer needs to itself be an asset). It is better to not mark extraneous objects as assets, since assets must be explicitly discarded, and only assets can own assets. Each of those requirements imposes a burden on the programmer. This burden can be helpful in detecting bugs, but should not be borne when not required. We changed the language so that `asset` can apply to individual states as well as entire contracts.

Third, the type system in Obsidian has significant implications on architecture. In a traditional object-oriented language, it is feasible to have many aliases to an object, with informal conventions regarding relationships between the object and the referencing objects. UML also distinguishes between composition, which implies ownership, and aggregation, which does not, reinforcing the idea that ownership in the sense in which Obsidian uses it is common and useful in typical object-oriented designs. Because of the use of ownership in Obsidian, using typestate with a design that does not express ownership sometimes requires refining the design so that it does. In this case study, we found applying ownership useful in refining our design. For example, when an insurance policy is purchased, the insurance service must hold the payout virtual currency until either the policy expires or it is paid. While the insurance service holds the currency, it must associate the currency for a policy with the policy itself. Does the policy, therefore, own the `Money`? If so, then what is the relationship between the client, who purchased the policy and has certain kinds of control over it, and the `Policy`, which cannot be held by the (untrusted) client? We resolved this question by adding a new object, the `PolicyRecord`. A `PolicyRecord`, which is itself Owned by the insurance service, has an Unowned reference to the `Policy` and an Owned reference to a `Money` object. This means that `PolicyRecord` is an `asset` when it is active (because it owns `Money`, which is itself an `asset`) but `Policy` does not need to be an asset. We found that thinking about ownership according to the Obsidian type system helped us refine and clarify our design. Without ownership, we might have chosen a less carefully considered design.

```
contract Policy {
  state Offered {
    int cost;
    int expirationTime;
  }

  state Active;
  state Expired;
  state Claimed;

  Policy@Offered(int c, int expiration) {
    ->Offered(cost = c, expirationTime = expiration);
  }

  transaction activate(Policy@Offered >> Active this) {
    ->Active;
  }


  transaction expire(Policy@Offered >> Expired this) {
    ->Expired;
  }
}
```

(a) Obsidian implementation of a Policy contract.

```
contract Policy {
  enum States {Offered, Active, Expired}
  States public currentState;
  uint public cost;
  uint public expirationTime;

  constructor (uint _cost, uint _et) public {
    cost = _cost;
    expirationTime = _et;
    currentState = States.Offered;
  }

  function activate() public {
    require(currentState == States.Offered,
      "Can't activate Policy not in Offered state.");
    currentState = States.Active;
    cost = 0;
    expirationTime = 0;
  }

  function expire() public {
    require(currentState == States.Offered,
      "Can't expire Policy not in Offered state.");
    currentState = States.Expired;
    cost = 0;
    expirationTime = 0;
  }
}
```

(b) Solidity implementation of a Policy contract.

Fig. 10. Comparison between Obsidian and Solidity implementations of a Policy contract from the insurance case study.

It is instructive to compare the Obsidian implementation to a partial Solidity implementation, which we wrote for comparison purposes. Figure 10 shows an example of why parts of the Obsidian implementation are substantially shorter. Note how the Solidity implementation requires repeated execution time tests to make sure each function only runs when the receiver is in the appropriate state. Obsidian code only invokes those transactions when the `Policy` object is in appropriate state; the runtime executes an equivalent dynamic check to ensure safety when the transactions are invoked from outside Obsidian code. Also, the Solidity implementation has `cost` and `expirationTime` fields in scope when inappropriate, so they need to be initialized repeatedly. In the Obsidian implementation, they are only set when the object is in the `Offered` state. Finally, the Solidity implementation must track the state manually via `currentState` and the `States` type, whereas this is done automatically in the Obsidian implementation. However, Solidity supports some features that are convenient and lead to more concise code: the Solidity compiler automatically generates getters for public fields, whereas Obsidian requires the user to write them manually, and built-in arrays can be convenient. However, the author of the Solidity implementation must be very careful to manage money manually; any money that is received by transactions must be accounted for, or the money will be stuck in the contract forever. Solidity also lacks a math library; completing the implementation would require us to provide our own square root function (which we use to compute distances).

We showed our implementation to our World Bank collaborators, and they agreed that it represents a promising design. There are various aspects of the full system that are not part of the case study, such as properly verifying cryptographic signatures of weather data, communicating with a real weather service and a real bank, and supporting multiple banks and insurers. However, in only a cursory review, one of the World Bank economists noticed a bug in the Obsidian code: the code always approved a claim requests even if the weather did not justify a claim according to

the policy's parameters. This brings to light two important observations. First, Obsidian, despite being a novel language, is readable enough to new users that they were able to understand the code. Second, type system-based approaches find particular classes of bugs, but other classes of bugs require either traditional approaches or formal verification to find.

8.2 Case Study 2: Shipping

8.2.1 Motivation. Supply chain tracking is one of the commonly proposed applications for blockchains [IBM 2019]. As such, we were interested in what implications Obsidian's design would have on an application that tracks shipments as they move through a supply chain. We collaborated with partners at IBM Research to conduct a case study of a simple shipping application. Our collaborators wrote most of the code, with occasional Obsidian help from us. We updated the implementation to use the polymorphic LinkedList contract, which became available only after the original implementation was done.

8.2.2 Results. The final implementation[2] consists of 141 non-comment, non-whitespace, non-printing lines of Obsidian code. We found it very encouraging that our collaborators were able to write the case study with relatively little input from us, especially considering that Obsidian is a research prototype that had extremely limited documentation at the time the case study was completed. Although this is smaller than the insurance case study, we noticed some interesting relationships between the Obsidian type system and object-oriented design.

Figure 11 summarizes an early design of the Shipping application.[3] Transactions can be invoked when a package changes status. For example, `depart` in `Transport` changes the state from `Load` to `InTransport`, creating a new `Leg` corresponding with the current step in the package's journey. However, the implementation does not compile; the compiler reports three problems. First, `LegList`'s `arrived` transaction attempts to invoke setArrival via a reference of type `Leg@Unowned`; this is disallowed, because setArrival changes the state of its receiver, which is unsafe through an `Unowned` reference. Second, `append` in `LegList` takes an `Unowned` leg to append, but uses it to transition to the `HasNext` state, which requires an `Owned` object. Third, `Transport`'s `depart` transaction attempts to append a new `Leg` to its `legList`. It does so by calling the `Leg` constructor, which takes a `Shared Transport`. But calling this constructor passing an owned reference (`this`) causes the caller's reference to become `Shared`, not `Owned`, which is inconsistent with the type of `depart`, which requires that `this` be owned (and specifically in state `InTransport`).

Figure 12 shows the final design of the application. This version passes the type checker. Note how a `LegList` contains only `Arrived` references to `Leg` objects. In addition, when a `Transport` is in `InTransit` state, it owns one `Leg`, which is also in `InTransit` state. Each `Leg` has an `Unowned` reference to its `Transport`, allowing the `TransportList` to own the `Transport`. A `TransportList` likewise only contains objects in `Unload` state; one `Transport` in `InTransport` state is referenced at the `Shipment` level.

We argue that although the type checker forced the programmer to revise the design, the revised design is better. In the first design, collections (`TransportList` and `LegList`) contain objects of dissimilar types. In the revised design, these collections contain only objects in the same state. This change is analogous to the difference between dynamically typed languages, such as LISP, in which collections may have objects of inconsistent type, and statically typed languages, such as Java, in which the programmer reaps benefits by making collections contain objects of consistent type. The typical benefit is that when one retrieves an object from the collection, there is no need to case-analyze on the element's type, since all of the elements have the same type. This means that

[2] https://github.com/laredo/Shipping.
[3] This version corresponds with git commit 8106e406e8ca005f8878dea5ac78e54b439fe509 in the Shipping repository.

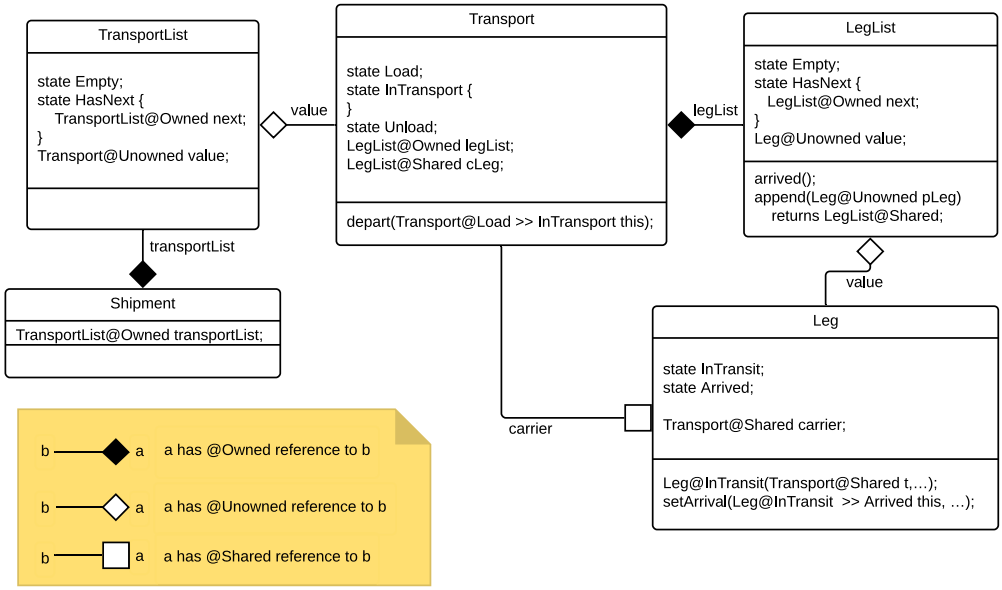

Fig. 11. Initial design of the Shipping application (which does not compile).

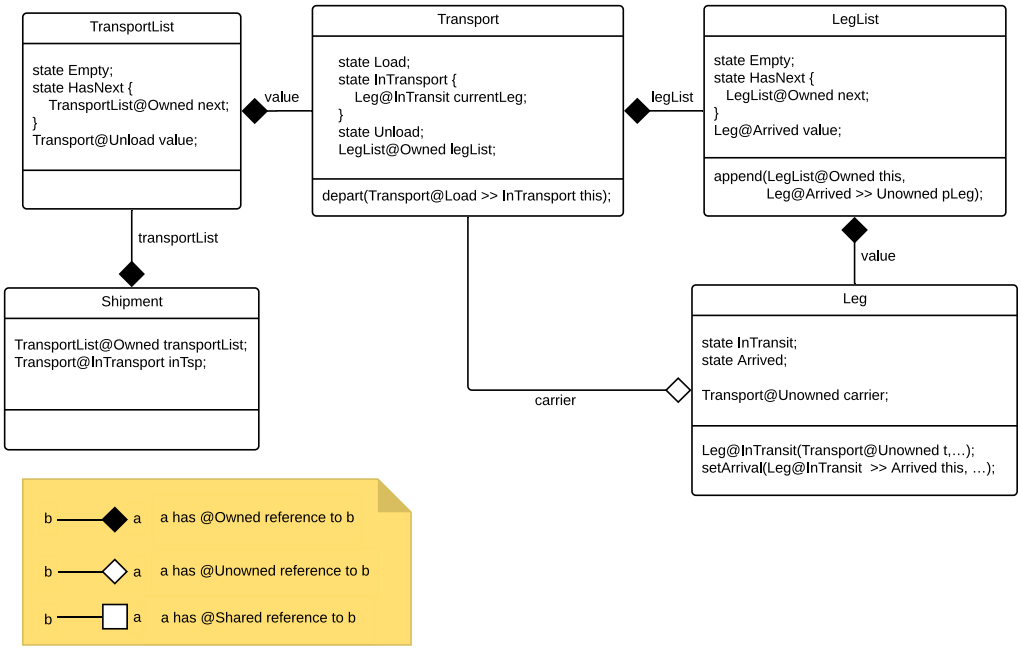

Fig. 12. Revised design of the Shipping application.

there can be no bugs that arise from neglecting to case-analyze, as can happen in the dynamically typed approach.

The revised version also reflects a better division of responsibilities among the components. For example, in the first version (Figure 11), `LegList` is responsible for both maintaining the list of

legs as well as recording when the first leg arrived. This violates the *single responsibility principle* [Martin et al. 2003]. In the revised version, `LegList` only maintains a list of `Leg` objects; updating their states is implemented elsewhere.

One difficulty we noticed in this case study, however, is that sometimes there is a conceptual gap between the relatively low-level error messages given by the compiler and the high-level design changes needed to improve the design. For example, the first error message in the initial version of the application shown in Figure 11 is as follows: `Cannot invoke setArrival on a receiver of type Leg@Owned; a receiver of type Leg@InTransit is required.` The programmer is required to figure out what changes need to be made; in this case, the `arrived` transaction should not be on `LegList`; instead, `LegList` should only include legs that are already in state `Arrived`. We hypothesize that more documentation and tooling may be helpful to encourage designers to choose designs that will be suitable for the Obsidian type system.

We also implemented a version of the Shipping application in Solidity, which required 197 non-comment, non-whitespace lines, so the Obsidian version (141 lines) took 72% as many lines as the Solidity version. The translation was straightforward; we translated each state precondition to an execution time assertion, and we flattened fields in states to the top (contract) level. Although the types no longer express the structural constraints that exist in the Obsidian version, the general structure of the code and data structures was identical except that we implemented containers with native Solidity arrays rather than linked lists. As with the prior case study, the additional length required for Solidity was generally due to execution time checks of properties that were established statically in the Obsidian version.

8.3 Case Study Summary

We asked four research questions above. We return to them now and summarize what we found.

(1) *RQ1: Does the aliasing structure in real blockchain applications allow use of ownership (and therefore typestate)?* The aliasing structure in the blockchain applications we implemented *does* allow use of ownership and typestate. However, it forces the programmer to carefully choose an ownership structure, rather than using ad hoc aliases. This can be restrictive but can also result in a simpler, cleaner design.
(2) *RQ2: Does typestate prevent the need for state checks and assertions?* Implementing smart contracts in Solidity typically requires a couple of lines of assertions for every function in smart contracts that are designed to use states. This makes the Obsidian code more concise, although some features that Obsidian currently lacks (such as auto-generated getters) improve concision in Solidity.
(3) *RQ3: Can realistic systems be built with Obsidian?* We and our collaborators were able to successfully build nontrivial smart contracts in Obsidian, despite the fact that Obsidian is a research prototype without much documentation.
(4) *RQ4: Do realistic systems have constructs that are naturally expressed as states and assets?* The applications that we chose benefited from representation with assets and states, since they represented objects of value and the transactions that were possible at any given time depended on the state of the object. Of course, not every application of smart contracts has this structure.

9 FUTURE WORK

Obsidian is a promising smart contract language, but it should not exist in isolation. Authors of applications for blockchain systems (known as *distributed applications*, or *Dapps*) need to be able to integrate smart contracts with front-end applications, such as web applications. Typically,

developers need to invoke smart contract transactions from JavaScript. We would like to build a mechanism for JavaScript applications to safely invoke transactions on Obsidian smart contracts. One possible approach is to embed Obsidian code in JavaScript to enable native interaction, coupled with a mapping between Obsidian objects and JSON.

Obsidian currently has limited IDE support. Although programmers can receive live feedback on errors while editing Obsidian code in our extension for Visual Studio Code, in the future, it would be better if there were debugger support so programmers could step through their Obsidian code in a debugger.

In the current implementation, Obsidian clients invoke all remote transactions sequentially. This means that another remote user might run intervening transactions, violating assumptions of the client program. More discussion of approaches to address this can be found in Section 5.2.

The type system-oriented approach in Obsidian is beneficial for many users, but it does not lead to verification of domain-specific program properties. In the future, it would be beneficial to augment Obsidian with a verification mechanism so that users can prove relevant properties of their programs formally. It would also be useful to conduct a corpus study to assess to what extent Obsidian's type system detects bugs that occur frequently in deployed code, and of those, which are exploitable security vulnerabilities.

The translation from Obsidian to Silica is presented informally. In the future, a formal definition of the translation would be beneficial.

One limitation of Obsidian's approach of using linear types is that it prevents assets from being lost but does not prevent them from becoming inaccessible. For example, one could write a contract that has a `deposit` operation but lacks a `withdraw` operation. Future analyses could try to show that there is always some execution path that could be used to withdraw assets.

Finally, Obsidian currently only supports Hyperledger Fabric. We would like to target Ethereum as well to demonstrate generality of the language as well as to enable more potential users to use the language.

10 CONCLUSIONS

With Obsidian we have shown how:

- Typestate can be combined with assets using a permissions system that expresses ownership to provide relevant safety properties for smart contracts, including asset retention.
- A unified approach for smart contracts and client programs can provide safety properties that cannot be provided using the approaches that are currently in use.
- A core calculus can encode key features of Obsidian and provide a sound foundation for the language.
- Applications can be built successfully with typestate and assets, with useful implications on architecture and object-oriented design.

Obsidian represents a promising approach for smart contract programming, including sound foundations and an implementation that enables real programs to execute on a blockchain platform. By formalizing useful safety guarantees and providing them in a programming language that was designed with user input, we hope to significantly improve safety of smart contracts that real programmers write. By combining techniques from human-computer interaction, traditional principles of type system design, and evaluation via case studies, we can obtain a language that is much better than if we used only one of those techniques alone.

APPENDIXES

A AUXILIARY JUDGEMENTS

A.1 Program Structure

We assume that the contracts and interfaces defined in a program are ambiently available via the *def* function, which retrieves the definition of a contract or interface (definition) by name. Likewise, the definition of a state S of contract or interface D can be retrieved via $sdef(D, S)$, and the definition of a transaction can be retrieved via $tdef(D, m)$. Note that for declaration variables $def(X)$ is the interface bound on X; similarly, $sdef(X, S)$ is the state in the bound on X. That is $sdef(X, S) = sdef(def(X), S)$.

Auxiliary Judgment 1: $stateFields(D, S)$

On individual states, *stateFields* gives only the fields defined directly in those states:

$$\frac{def(C) = \text{ contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\ \{\overline{ST}\ \overline{M}\} \quad S\ \overline{F} \in \overline{ST}}{stateFields(C, S) = \overline{F}} \qquad \frac{}{stateFields(I, S) = \cdot}$$

Auxiliary Judgment 2: $unionFields(T)$

The *unionFields* function looks up the fields that are defined in ANY of the states in a set of states. Note that the syntax guarantees that any field has consistent types in all states in which it is defined. This is useful when it is known that one of two different types captures the state of an object, but it is not known which one.

$$\frac{F = \cup_{S \in \overline{S}} stateFields(D, S)}{unionFields(D@\overline{S}) = F} \qquad \frac{\begin{array}{c} T_{ST} \in \{\text{Shared, Owned, Unowned}\} \\ cdef(C) = \text{ contract } C\{\overline{[\text{asset}]S\ F_S}\ \overline{M}\} \\ F = \cup_{S \in \overline{F_S}} stateFields(C, S) \end{array}}{unionFields(D@T_{ST}) = F}$$

Auxiliary Judgment 3: $intersectFields(T)$

The *intersectFields* function looks up the fields that are defined in ALL of the states in a set of states. Note that the syntax guarantees that any field has consistent types in all states in which it is defined.

$$\frac{F = \cap_{S \in \overline{S}} stateFields(D, S)}{intersectFields(D@\overline{S}) = F} \qquad \frac{\begin{array}{c} T_{ST} \in \{\text{Shared, Owned, Unowned}\} \\ cdef(C) = \text{contract } C\{\overline{[\text{asset}]S\ F_S}\ \overline{M}\} \\ F = \cap_{S \in \overline{F_S}} stateFields(D, S) \end{array}}{intersectFields(C@T_{ST}) = F}$$

Auxiliary Judgment 4: $contract(T_C)$

The *contract* function relates types with their contracts.

$$\frac{}{contract(T_C@T_{ST}) = T_C}$$

Auxiliary Judgment 5: $contractFields(C)$

On contracts, contractFields gives the set of field declarations defined in all of a contract's states,

$$contractFields(C) \triangleq intersectFields(C@\text{Unowned}).$$

Auxiliary Judgment 6: $fieldTypes_s(\Delta; \overline{T_{fs}\ f_s})$

$fieldTypes$ gives the current types of the fields, given that some of them may be overridden in the current context.

$$\frac{}{fieldTypes_s(\cdot; \overline{T_{fs}\ f_s}) = \overline{T_{fs}}} \qquad \frac{f \in \overline{f_s} \qquad fieldTypes_s(\Delta; \overline{T_{fs}\ f_s}) = \overline{T'}}{fieldTypes_s(\Delta, s.f : T; \overline{T_{fs}\ f_s}) = T, \overline{T'}}$$

$$\frac{}{fieldTypes_s(\Delta, b : T; \overline{T_{fs}\ f_s}) = fieldTypes_s(\Delta; \overline{T_{fs}\ f_s})}$$

Auxiliary Judgment 7: $transactionName(M)$, $transactionName(M_{SIG})$, $transactionNames(\overline{M})$

$$transactionName\left(Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})T_{ST} \gg T_{ST}\right) \triangleq m$$
$$transactionName\left(Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})T_{ST} \gg T_{ST}\ e\right) \triangleq m$$
$$transactionName\left(\overline{T_{ST} >> T_{ST} f} Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})T_{ST} \gg T_{ST}\right) \triangleq m$$
$$transactionName\left(\overline{T_{ST} >> T_{ST} f} Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})T_{ST} \gg T_{ST}\ e\right) \triangleq m$$

$$transactionNames\left(\overline{M}\right) \triangleq \overline{transactionName\,(M)}$$

Auxiliary Judgment 8: $states\,(D)$

The *states* function extracts the state definitions from contracts and interfaces.

$$\frac{\text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\{\overline{ST}\ \overline{M}\}}{states(C) = \overline{ST}} \qquad \frac{\text{interface } I\langle\overline{T_G}\rangle\{\overline{ST}\ \overline{M_{SIG}}\}}{states(I) = \overline{ST}}$$

Auxiliary Judgment 9: $stateName(S)$, $stateNames(D)$,

The *stateNames* function extracts the names of states from declarations by iteratively applying $stateName$ to individual states.

$$\frac{}{stateName([\text{asset}]\ S\ \overline{F}) = S} \qquad \frac{states(D) = \overline{S}}{stateNames(D) = \overline{stateName(S)}}$$

Auxiliary Judgment 10: $params\,(D)$, $params\,(M)$

The *params* function extracts the type parameters from declarations.

$$\frac{def(C) = \text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\{\overline{ST}\ \overline{M}\}}{params(C) = \overline{T_G}} \qquad \frac{def(I) = \text{interface } I\langle\overline{T_G}\rangle\{\overline{ST}\ \overline{M_{SIG}}\}}{params(I) = \overline{T_G}}$$

$$\frac{}{params(T\ m\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})\ T_{ST_i} \gg T_{ST_f}\ e) = \overline{T_M}}$$

$$\frac{}{params(\overline{T_{STs1} >> T_{STs2} f}\ T\ m\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}\ x})\ T_{ST_i} \gg T_{ST_f}\ e) = \overline{T_M}}$$

Auxiliary Judgment 11: $T_1 \approx T_2$ **Ownership equality**

$$\frac{}{T_1 \approx T_1}\ \approx\text{-Refl} \qquad \frac{T_1 \approx T_2}{T_2 \approx T_1}\ \approx\text{-Sym} \qquad \frac{T_1 \approx T_2 \qquad T_2 \approx T_3}{T_1 \approx T_3}\ \approx\text{-Trans}$$

$$\frac{maybeOwned(T_1) \qquad maybeOwned(T_2)}{T_1 \approx T_2}\ \approx\text{-O-O} \qquad \frac{notOwned(T_1) \qquad notOwned(T_2)}{T_1 \approx T_2}\ \approx\text{-U-U}$$

Auxiliary Judgment 12: $\overline{S}$ **ok** **Well-formed state sequence**

Well-formed states cannot have conflicts regarding ownership, and if any states are specified, then Owned would be redundant. There must be no duplicates in the list.

$$\frac{\overline{S}\ \textbf{statename-list}}{\overline{S}\ \textbf{ok}}$$

Auxiliary Judgment 13: $\overline{S}$ **statename-list** **Well-formed statename sequence**

$$\frac{}{S\ \textbf{statename-list}} \qquad \frac{\overline{S}\ \textbf{statename-list} \qquad S' \notin \overline{S}}{\overline{S}, S'\ \textbf{statename-list}}$$

Auxiliary Judgment 14: T **ok** **Well-formed type**

$$\frac{\overline{S}\ \textbf{ok}}{T_C@\overline{S}\ \textbf{ok}} \qquad \frac{}{T_C@p\ \textbf{ok}} \qquad \frac{}{\mathsf{unit}\ \textbf{ok}}$$

A.2 Reasoning About Types

Auxiliary Judgment 15: *toPermission* (T_{ST})

The judgment *toPermission* provides a conservative approximation of ownership to ensure that if *toPermission* indicates non-ownership, the type is definitely disposable.

$$toPermission\left(\overline{S}\right) \triangleq \mathsf{Owned} \qquad toPermission\,(\mathsf{Unowned}) \triangleq \mathsf{Unowned}$$
$$toPermission\,(p) \triangleq \mathsf{Owned} \qquad toPermission\,(\mathsf{Shared}) \triangleq \mathsf{Shared}$$
$$toPermission\,(\mathsf{Owned}) \triangleq \mathsf{Owned}$$

Auxiliary Judgment 16: $possibleStates_\Gamma\,(T_C@T_{ST}) = T_{ST}$

$$\frac{}{possibleStates_\Gamma(T_C@\overline{S}) = \overline{S}} \qquad \frac{P \in \{\mathsf{Owned}, \mathsf{Shared}, \mathsf{Unowned}\}}{possibleStates_\Gamma(T_C@P) = stateNames(def\,(T_C))}$$

$$\frac{[\mathsf{asset}]\ X@p\ \mathsf{implements}\ I\langle\overline{T}\rangle@T_{ST} \in \Gamma}{possibleStates_\Gamma(T_C@p) = possibleStates_\Gamma(T_C@T_{ST})}$$

Auxiliary Judgment 17: $\Gamma \vdash isAsset\,(\mathbf{T})$

$$\frac{\mathsf{asset}\ S\ \overline{F} \in possibleStates_\Gamma(D\langle\overline{T}\rangle@T_{ST})}{\Gamma \vdash isAsset(D\langle\overline{T}\rangle@T_{ST})} \qquad \frac{\mathsf{asset}\ X@p\ \mathsf{implements}\ I\langle\overline{T}\rangle@T_{ST_i} \in \Gamma}{\Gamma \vdash isAsset(X@T_{ST})}$$

Auxiliary Judgment 18: $\Gamma \vdash \mathit{nonAssetState}$ (**ST**)

$$\frac{}{\Gamma \vdash \mathit{nonAssetState}(S\ \overline{F})}$$

Auxiliary Judgment 19: $\Gamma \vdash \mathit{nonAsset}$ (**T**)

$$\frac{\Gamma \vdash \mathit{nonAssetState}(\mathit{possibleStates}_\Gamma(D\langle\overline{T}\rangle@T_{ST}))}{\Gamma \vdash \mathit{nonAsset}(D\langle\overline{T}\rangle@T_{ST})} \qquad \frac{X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST_i} \in \Gamma}{\Gamma \vdash \mathit{nonAsset}(X@T_{ST})}$$

Auxiliary Judgment 20: $\Gamma \vdash \mathit{disposable}$ (**T**)

The *disposable* judgement describes reference types that are NOT owning references to assets. When applied to a set of states, all states must be disposable in order for the set to be disposable.

$$\frac{\mathit{notOwned}(T)}{\Gamma \vdash \mathit{disposable}(T_C@T_{ST})} \qquad \frac{\mathit{maybeOwned}(T_C@T_{ST}) \qquad \Gamma \vdash \mathit{nonAsset}(T_C@T_{ST})}{\Gamma \vdash \mathit{disposable}(T_C@T_{ST})}$$

Auxiliary Judgment 21: *notOwned* (**T**)

$$\frac{}{\mathit{notOwned}(T_C@\mathsf{Unowned})} \qquad \frac{}{\mathit{notOwned}(T_C@\mathsf{Shared})} \qquad \frac{}{\mathit{notOwned}(\mathsf{unit})}$$

Auxiliary Judgment 22: *maybeOwned* (**T**)

$$\frac{T_{ST} <:_* \mathsf{Owned}}{\mathit{maybeOwned}(T_C@T_{ST})} \qquad \frac{}{\mathit{maybeOwned}(T_C@p)}$$

Note that all permission variables could be owned, because we only have upper bounds on permissions. Therefore, we must treat all permission variables as though they may be owned.

Auxiliary Judgment 23: $\Gamma \vdash \mathit{bound}\ (T)$

The bound of a type T or permission is the most specific concrete (i.e., non-parametric) type that is a supertype of T. Likewise, the bound of a state T_{ST} is the most specific state that is a supertype of T_{ST}. For example, if we know from a type parameter that the type variable X must implement an interface $I\langle\overline{T}\rangle$ and p must be a subpermission of Owned, then the bound of $X@p$ is $I\langle\overline{T}\rangle$.Owned. However, a concrete type such as $C\langle\overline{T}\rangle@S$ is already as specific as possible—therefore, its bound is itself.

$$\frac{}{\Gamma \vdash \mathit{bound}(\mathsf{unit}) = \mathsf{unit}} \qquad \frac{\Gamma \vdash \mathit{bound}_*(T_{ST}) = T'_{ST}}{\Gamma \vdash \mathit{bound}(D\langle\overline{T}\rangle@T_{ST}) = D\langle\overline{T}\rangle@T'_{ST}}$$

$$\frac{[\mathsf{asset}]\ X@p \text{ implements } I\langle\overline{T}\rangle@T'_{ST} \in \Gamma \qquad \Gamma \vdash \mathit{bound}_*(T_{ST}) = T'_{ST}}{\Gamma \vdash \mathit{bound}(X@T_{ST}) = I\langle\overline{T}\rangle@T'_{ST}}$$

Auxiliary Judgment 24: $\Gamma \vdash bound_*(T_{ST})$

$$\frac{P \in \{\text{Owned}, \text{Shared}, \text{Unowned}\}}{\Gamma \vdash bound_*(P) = P} \qquad \frac{}{\Gamma \vdash bound_*(\overline{S}) = \overline{S}}$$

$$\frac{[\text{asset}]\ X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST} \in \Gamma}{\Gamma \vdash bound_*(p) = T_{ST}}$$

Auxiliary Judgment 25: $nonVar(T), nonVar(T_C), nonVar(T_{ST})$

$$\frac{}{nonVar(D\langle\overline{T}\rangle@T_{ST})} \qquad \frac{}{nonVar(\text{unit})} \qquad \frac{}{nonVar(D\langle\overline{T}\rangle)}$$

$$\frac{T_{ST} \in \{\text{Owned}, \text{Shared}, \text{Unowned}\}}{nonVar(T_{ST})} \qquad \frac{}{nonVar(\overline{S})}$$

Auxiliary Judgment 26: $isVar(T), isVar(T_C), isVar(T_{ST})$

$$\frac{}{isVar(X@T_{ST})} \qquad \frac{}{isVar(X)} \qquad \frac{}{isVar(p)}$$

Auxiliary Judgment 27: $Var(T_G), PermVar(T_G), Perm(T)$

$$Var([\text{asset}]\ X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST}) \triangleq X$$
$$PermVar([\text{asset}]\ X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST}) \triangleq p$$

$$Perm(T_C@T_{ST}) \triangleq T_{ST}$$
$$Perm(\text{unit}) \triangleq \text{Unowned}$$

Auxiliary Judgment 28: $implementOk_\Gamma(I\langle\overline{T}\rangle, M_{SIG}), implementOk_\Gamma(I\langle\overline{T}\rangle, ST)$

$$\frac{\begin{array}{c} specializeTrans_\Gamma(m, I\langle\overline{T}\rangle) = T'_{ret}\ m\langle\overline{T'_M}\rangle(\overline{T' \gg T'_{ST}}\ x)\ T'_{ST_i} \gg T'_{ST_f} \\ \overline{\Gamma \vdash T' <: T} \qquad \overline{\Gamma \vdash T_{ST} <:_* T'_{ST}} \qquad \Gamma \vdash T'_{ST_i} <:_* T_{ST_i} \\ \Gamma \vdash T_{ST_f} <:_* T'_{ST_f} \qquad \Gamma \vdash T_{ret} <: T'_{ret} \end{array}}{implementOk_\Gamma(I\langle\overline{T}\rangle, T_{ret}\ m\langle\overline{T_M}\rangle(\overline{T \gg T_{ST}}\ x)\ T_{ST_i} \gg T_{ST_f})}$$

$$\frac{sdef(S, I\langle\overline{T}\rangle) = \text{asset } S}{implementOk_\Gamma(I\langle\overline{T}\rangle, [\text{asset}]\ S\ \overline{F})} \qquad \frac{sdef(S, I\langle\overline{T}\rangle) = S}{implementOk_\Gamma(I\langle\overline{T}\rangle, S\ \overline{F})}$$

To check $implementOk_\Gamma\left(I\langle\overline{T}\rangle, S\right)$, we only need to ensure that if our state is an asset, then the state we are implementing is also an asset.

Auxiliary Judgment 29: $subsOk_\Gamma\ (T, T_G)$

$$\frac{\Gamma \vdash D\langle\overline{T_1}\rangle@T_{ST} <: I\langle\overline{T_2}\rangle@T'_{ST}}{subsOk_\Gamma(D\langle\overline{T_1}\rangle@T_{ST}, \text{asset } X@p \text{ implements } I\langle\overline{T_2}\rangle@T'_{ST})}$$

$$\frac{\Gamma \vdash D\langle\overline{T_1}\rangle@T_{ST} <: I\langle\overline{T_2}\rangle@T'_{ST} \qquad \Gamma \vdash nonAsset(D\langle\overline{T_1}\rangle.\text{Owned})}{subsOk_\Gamma(D\langle\overline{T_1}\rangle@T_{ST},\ X@p \text{ implements } I\langle\overline{T_2}\rangle@T'_{ST})}$$

We can substitute a non-asset for an asset generic parameter, but not vice versa. Note that, as we can use type variables without their corresponding permission variable (e.g., we can write X@Owned, not just $X@p$), we must check whether the generic parameter is an asset in *any* state, not just its bound. Similarly, we must check if the type we pass is an asset in *any* state, not just the one we pass.

Auxiliary Judgment 30: $genericsOk_\Gamma\ (T_G)$

$genericsOk_\Gamma\ (T_G)$ expresses whether a use of a type parameter is suitable when the parameter must implement a particular interface.

$$\frac{\begin{array}{c}\forall T \in \overline{T}, isVar(T) \implies T \in Var(\Gamma) \qquad \overline{subsOk_\Gamma(T, params(I))} \qquad \Gamma \vdash nonAsset(I\langle\overline{T}\rangle.\text{Owned}) \\ \forall T_G \in \Gamma, (Var(T_G) = X \text{ or } PermVar(T_G) = p) \implies T_G = X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST} \\ T_{ST} = \overline{S} \implies \forall S \in \overline{S}, S \in stateNames(I)\end{array}}{genericsOk_\Gamma(\ X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST})}$$

$$\frac{\begin{array}{c}\forall T \in \overline{T}, isVar(T) \implies T \in Var(\Gamma) \qquad \overline{subsOk_\Gamma(T, params(I))} \\ \forall T_G \in \Gamma, (Var(T_G) = X \text{ or } PermVar(T_G) = p) \implies T_G = X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST} \\ T_{ST} = \overline{S} \implies \forall S \in \overline{S}, S \in stateNames(I)\end{array}}{genericsOk_\Gamma(\text{asset } X@p \text{ implements } I\langle\overline{T}\rangle@T_{ST})}$$

Auxiliary Judgment 31: $\sigma\ (T/T_G)\ (e)$

$\sigma\ (T/T_G)\ (e)$ defines how to substitute a concrete type for a type parameter.

$$\frac{T_G = [\text{asset}]\ X@p \text{ implements } I\langle\overline{T_2}\rangle@T'_{ST}}{\sigma(D\langle\overline{T}\rangle@T_{ST}/T_G)(e) = [D\langle\overline{T}\rangle/X][T_{ST}/p]e}$$

$$\frac{\overline{T} = T_1, T_2, \ldots, T_n \qquad \overline{T_G} = T_{G_1}, T_{G_2}, \ldots, T_{G_n}}{\sigma(\overline{T/T_G})(e) = (\sigma(T_n/T_{G_n}) \circ \sigma(T_{n-1}/T_{G_{n-1}}) \circ \cdots \circ \sigma(T_1/T_{G_1}))\ (e)}$$

Auxiliary Judgment 32: $specializeTrans_\Gamma\left(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle\right)$

$specializeTrans_\Gamma(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle))$ defines how to specialize a transaction definition by substituting concrete types for the transaction's type parameters $\overline{T_M}$ as well as for the type parameters on the receiver's contract $\overline{T_G}$.

$$\frac{tdef(D, m) = M \qquad \overline{T_M} = params(M) \qquad \overline{T_G} = params(D) \qquad \overline{subsOk_\Gamma(T, T_G)} \qquad \overline{subsOk_\Gamma(T_2, T_M)}}{specializeTrans_\Gamma(m\langle\overline{T_2}\rangle, D\langle\overline{T}\rangle) = \sigma(\overline{T_2/T_M})(\sigma(\overline{T/T_G})(M))}$$

Auxiliary Judgment 33: $merge(\Delta, \Delta') = \Delta''$

The *merge* function computes a new context from contexts that resulted from branching. It ensures that ownership is consistent across both branches and takes the union of state sets for each variable.

For brevity, let $d ::= x \mid x.f$.

$$\frac{merge(\Delta; \Delta') = \Delta''}{merge(\Delta'; \Delta) = \Delta''}\ \textsc{Sym} \qquad \frac{merge(\Delta; \Delta') = \Delta''}{merge(\Delta, d : T; \Delta', d : T') = \Delta'', d : (T \oplus T')}\ \oplus$$

$$\frac{x \notin Dom(\Delta') \qquad merge(\Delta, \Delta') = \Delta'' \qquad \Gamma \vdash disposable(T)}{merge(\Delta, x : T; \Delta') = \Delta''}\ \textsc{Dispose-disposable}$$

$T \oplus T \triangleq T$

$T_C@\text{Owned} \oplus T_C@\overline{S} \triangleq T_C@\text{Owned}$

$T_C@\text{Shared} \oplus T_C@\text{Unowned} \triangleq T_C@\text{Unowned}$

$T_C@\overline{S} \oplus T_C@\overline{S'} \triangleq T_C@(S \cup S')$

$C\langle\overline{T}\rangle@T_{ST} \oplus I\langle\overline{T}\rangle@T'_{ST} \triangleq I\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$ if $def(C) = \text{contract } C\langle\overline{T_G}\rangle \text{ implements } I\langle\overline{T}\rangle\{\ldots\}$

$D\langle\overline{T}\rangle@T_{ST} \oplus D\langle\overline{T}\rangle@T'_{ST} \triangleq D\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$.

Auxiliary Judgment 34: $funcArg(T_C@T_{STpassed}, T_C@T_{STinput-decl}, T_C@T_{SToutput-decl})$

This function specifies the output permission for a function argument that started with a particular permission and was passed to a formal parameter with given initial and final permission specifications. The function is only defined for inputs that correspond with well-typed invocations.

$$\frac{maybeOwned(T_C@T_{STpassed})}{funcArg(T_C@T_{STpassed}, T_C@\text{Unowned}, T_C@T_{SToutput-decl}) = T_C@T_{STpassed}}\ \textsc{funcArg-owned-unowned}$$

$$\frac{}{funcArg(T_C@\text{Shared}, T_C@\text{Unowned}, T_C@T_{SToutput-decl}) = T_C@T_{\text{Shared}}}\ \textsc{funcArg-shared-unowned}$$

$$\frac{T_C@T_{STinput-decl} \neq \text{Unowned}}{funcArg(T_C@T_{STpassed}, T_C@T_{STinput-decl}, T_C@T_{SToutput-decl}) = T_C@T_{SToutput-decl}}\ \textsc{funcArg-other}$$

Auxiliary Judgment 35: $funcArgResidual(T_C@T_{STpassed}, T_C@T_{STinput-decl}, T_C@T_{SToutput-decl})$

This function specifies the type of the reference that remains after an argument is passed to a function.

$$\frac{maybeOwned(T_C@T_{STpassed})}{funcArgResidual(T_C@T_{STpassed}, T_C@\text{Unowned}, T_C@T_{SToutput-decl}) = T_C@T_{STpassed}}\ \text{FAR-OU}$$

$$\frac{}{funcArgResidual(T_C@\text{Shared}, T_C@\text{Unowned}, T_C@T_{SToutput-decl}) = T_C@T_{\text{Shared}}}\ \text{FAR-SU}$$

$$\frac{T_C@T_{STinput-decl} \neq \text{Unowned}}{funcArgResidual(T_C@T_{STpassed}, T_C@T_{STinput-decl}, T_C@T_{SToutput-decl}) = T_C@\text{Unowned}}\ \text{FAR-*}$$

B TRANSLATION FROM OBSIDIAN TO SILICA

The implementation of Obsidian implements type checking directly on Obsidian programs rather than first translating to Silica. However, we define the semantics of Obsidian by translation to Silica. The $\leadsto$ relation operates in a variable binding environment Δ that maps variables to their types, as well as the type bounds context Γ. Because Obsidian supports declaring local variables with only contracts specified, not full types (whereas Silica has no variable declarations at all, only let-bindings), we need an additional context, Θ, which maps local variable names to declared contract names:

$$\Theta ::= \overline{x : C}.$$

When translating assignments, the translator checks to make sure the assigned expression's contract is consistent with the contract that was specified in the variable's declaration. This contrasts with Δ, which tracks complete types (not just contract names) of variables. Δ is needed for inferring types in translated expressions. After declaring local variable x but before an assignment to x, x is recorded in Θ but not Δ. After the assignment, the variable is in both contexts.

The translation from Obsidian to Silica can be undefined due to type errors in the source Obsidian program. For brevity, we only show a few example rules to summarize how the translation works. The sequence operator is right-associative.

The variable name _ indicates that the translator chooses a fresh variable name, which has not previously been used.

$\boxed{\Gamma; \Delta; \Theta \vdash_s \overline{stmt} \leadsto e \dashv \Delta'}$

$$\frac{}{\Gamma; \Delta; \Theta \vdash_s \bullet \leadsto () \dashv \Delta} \textsc{Empty}$$

When translating an assignment, because Silica requires A-normal form, we let-bind the right-hand-side to a fresh variable, x', so that we can translate the assignment to a Silica assignment. Assignment requires that the left-hand-side variable has already been declared.

$$\frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash_s e \leadsto e' \dashv \Delta' \qquad \Gamma; \Delta \vdash_s e' : C@T_{ST} \dashv \Delta^* \qquad x : C \in \Theta \\ \Gamma'; \Delta', x : C@T_{ST}; \Theta \vdash_s \overline{stmts} \leadsto e'' \dashv \Delta'' \qquad x' \text{ fresh}\end{array}}{\Gamma; \Delta; \Theta \vdash_s x = e; \overline{stmts} \leadsto \mathsf{let}\ x' : C@T_{ST} = e'\ \mathsf{in}\ \mathsf{let}\ \_ : \mathsf{unit} = x := x'\ \mathsf{in}\ e'' \dashv \Delta''} \textsc{Assign}$$

In a declaration without an assignment, we need only record in Θ the declared contract name for future checking.

$$\frac{x \notin \mathrm{dom}(\Theta) \qquad \Gamma; \Delta; \Theta, x : C \vdash_s \overline{stmts} \leadsto e \dashv \Delta'}{\Gamma; \Delta; \Theta \vdash_s C\ x; \overline{stmts} \leadsto e \dashv \Delta'} \textsc{Decl}$$

In a declaration *with* an assignment, we compute the type of e to prepare the appropriate typing context for the rest of the statements.

$$\frac{\begin{array}{c}x \notin \mathrm{dom}(\Theta) \qquad \Gamma; \Delta'; \Theta, x : C \vdash_s e \leadsto e' \dashv \Delta'' \\ \Gamma; \Delta \vdash_s e' : C@T_{ST} \dashv \Delta^* \\ \Gamma; \Delta', x : C@T_{ST}; \Theta, x : C \vdash_s \overline{stmts} \leadsto e'' \dashv \Delta''\end{array}}{\Gamma; \Delta; \Theta \vdash_s C\ x = e; \overline{stmts} \leadsto \mathsf{let}\ x : C@T_{ST} = e'\ \mathsf{in}\ e'' \dashv \Delta''} \textsc{DeclAssign}$$

The overline in the premise of Switch indicates that each statement is translated to a Silica expression e_i''. The ellipsis here represents nested generation of dynamic state checks according to

the same structure shown for the first one, each using the appropriate e_i''. The pattern concludes with revert since switch requires that one of the cases matches.

$$\frac{\begin{array}{c}\Gamma;\Delta;\Theta \vdash_s e \rightsquigarrow e' \dashv \Delta' \qquad \overline{\Gamma;\Delta';\Theta \vdash_s stmt \rightsquigarrow e'' \dashv \Delta''} \\ \Gamma;\Delta \vdash_s e : C@T_{ST} \dashv \Delta^* \qquad P = toPermission(T_{ST}) \qquad x \text{ fresh} \\ BR = \text{if } x \text{ in}_P\ S_1 \text{ then } e_1'' \text{ else } \ldots \text{ else revert} \\ \Gamma;\Delta'';\Theta \vdash_s \overline{stmts} \rightsquigarrow stmts' \dashv \Delta''' \\ NL = \text{let } x : C@T_{ST} = e' \text{ in } BR\end{array}}{\Gamma;\Delta;\Theta \vdash_s \text{switch } e\{\overline{\text{case } S\{\overline{stmt}\}}\}; \overline{stmts} \rightsquigarrow \text{let } \_ : \text{unit} = NL \text{ in } stmts' \dashv \Delta'''}\ \textsc{Switch}$$

Expressions of the form `S::f` are translated to bound variables. We use the convention that no variable names in the source program may begin with `__` to avoid collisions.

$$\frac{}{\Gamma;\Delta;\Theta \vdash_s S :: x \rightsquigarrow \_\_S\_\_x \dashv \Delta}\ \textsc{S::x}$$

C SOUNDNESS THEOREMS

Theorem C.1 (Progress). *If e is a closed expression and $\Gamma;\Delta \vdash_s e : T \dashv \Delta'$, then at least one of the following holds:*

1. *e is a value.*
2. *For any environment Σ such that Γ, Σ, Δ **ok**, $\Sigma, e \rightarrow \Sigma', e'$ for some environment Σ'.*
3. *e is stuck at a bad state transition—that is, $e = \mathbb{E}[l \rightarrow_{Shared} S(\overline{s})]$ where $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S'(\ldots)$, $S \neq S'$, $\rho(l) \in \phi$, and $\Gamma;\Delta \vdash_s l : C\langle\overline{T'}\rangle@Shared \dashv \Delta'$.*
4. *e is stuck at a reentrant invocation – that is, $e = \mathbb{E}[l.m(\overline{s})]$ where $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\ldots)$, $\rho(l) \in \psi$.*
5. *e is stuck in a nested dynamic state check – that is, $e = \mathbb{E}[\text{if } s \text{ in}_{shared}\ T_{ST} \text{ then } e_1 \text{ else } e_2]$ where $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$ and $\rho(l) \in \phi$.*

Proof. By induction on the derivation of $\Gamma;\Delta \vdash_s e : T \dashv \Delta'$.

Case: T-lookup. $e = b$. We case-analyze on b.
- **Subcase: $b = x$.** Then b is not closed. Contradiction.
- **Subcase: $b = l$.** Suppose Γ, Σ, Δ ok. By global consistency, $l \in dom(\Sigma_\rho)$. Then $b \rightarrow \Sigma_\rho(l)$ by rule E-lookup.
- **Subcase: $b = o$.** Then b is a value.

Case: T-let. Because e is closed, $e = \text{let } x : T = e_1 \text{ in } e_2$. Otherwise, since e is closed, e_1 is closed, and the induction hypothesis applies to e_1. This leaves several cases:
- **Case: e_1 is a value v** The properties of the context permit creating a fresh indirect reference l that is not in ρ. By E-let, $\Sigma, \text{let } x : T = v \text{ in } e \rightarrow [\rho[l \mapsto v]/\rho]\,\Sigma, [l/x]e$.
- **Case: $\Sigma, e_1 \rightarrow \Sigma', e_1'$.** Then E-letCongr applies, and $\Sigma, e \rightarrow \Sigma', \text{let } x : T = e_1' \text{ in } e_2$.
- **Case: e_1 is stuck** with $e_1 = \mathbb{E}[l \rightarrow_{Shared} S(\overline{s})]$. Then

$$e = \text{let } x : T = \mathbb{E}[l \rightarrow_{Shared} S(\overline{s})] \text{ in } e_2$$
$$e = \mathbb{E}'[l \rightarrow_{Shared} S(\overline{s})].$$

- **Case: e_1 is stuck** with $e_1 = \mathbb{E}[\mathbb{E}[l.m(\overline{s})]$. Then

$$e = \text{let } x : T = \mathbb{E}[\mathbb{E}[l.m(\overline{s})] \text{ in } e_2$$
$$e = \mathbb{E}'[\mathbb{E}[l.m(\overline{s})].$$

Case: T-assign. Because e is closed, $e = l' := l''$. By memory consistency, $l' \in dom(\rho)$. By E-assign, $\Sigma, l := l' \rightarrow [\rho[l \mapsto o]/\rho]\,\Sigma, ()$.

Case: T-new. Because e is closed, $e = \text{new } C\langle\overline{T}\rangle@S(\overline{l})$ (any variables x would be free, so all parameters must be locations). The properties of the context permit creating a fresh object reference o that is not in μ. $\overline{l}$ are a free locations of e, so by memory consistency (6.2), $\overline{l} \in dom(\rho)$, and $\overline{\rho(l)}$ is well-defined. By E-new:

$$\Sigma, \text{new } C\langle\overline{T}\rangle@S(\overline{l}) \rightarrow [\mu[o \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l)})]/\mu]\,\Sigma, o.$$

Case: T-this-field-def. Because e is closed, $e = l.f_i$. By assumption:
(1) $\Gamma; \Delta \vdash_l l.f_i : T \dashv \Delta'$.
(2) Γ, Σ, Δ **ok**.
By memory consistency, $\rho(l) = o$ for some o and $\mu(o) = C\langle\overline{T'}\rangle@S(\overline{s'})$. Note that $1 \leq i \leq |\overline{s'}|$ by well-typedness of $s.f_i$ and global consistency. By rule E-field, $\Sigma, s.f_i \rightarrow \Sigma, s'_i$.

Case: T-this-field-ctxt. Identical to the *This-field-def* case.

Case: T-field-update. Because e is closed, $e = l.f_i := l'$. By memory consistency, $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\overline{l''})$. $fields(C\langle\overline{T'}\rangle@S)$ is ambiently available. By E-fieldUpdate, $\Sigma, l.f_i := l' \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(l''_1, l''_2, \ldots, l''_{i-1}, l', l''_{i+1}, \ldots, l''_{|l''|})]/\mu]\,\Sigma, ()$.

Case: T-inv. Because e is closed, $e = l_1.m\langle\overline{T}\rangle(\overline{l_2})$. By memory consistency, $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\ldots)$. The transaction is ambiently available. We generate fresh l'_1 and l'_2 so that they are not in $dom(\rho)$. If $rho(l_1) \in \psi$, then e is stuck at a reentrant invocation. Otherwise, let
(1) $\Sigma' = \Sigma[l'_1 \mapsto \rho(l_1)][\overline{l'_2 \mapsto \rho(l_2)}]$.
(2) $\xi' = \overline{PermVar(T_D) \mapsto Perm(T)}, \overline{PermVar(T_G) \mapsto Perm(T_M)}$.
(3) $\Sigma'' = [\xi'/\xi]\,[\psi, \rho(l_1)/\psi]\,\Sigma'$.
(4) $e' = tdef(C, m)$.
Then by E-Inv, $\Sigma, e \rightarrow \Sigma'', \boxed{\overline{[l'_1/x]}[l'_2/\text{this}]e'}^{\rho(l_1)}$

Case: T-privInv. Analogous to the *Public-Invoke* case, using rule E-Inv-Private, except that the invocation is never stuck (E-Inv-Private does not check that $rho(l_1) \notin \psi$).

Case: T-$\rightarrow_p$. Because e is closed, $e = l \rightarrow_p S(\overline{l'})$. By assumption, $l : C\langle\overline{T'}\rangle@T_{ST}$. By memory consistency, $l \in dom(\rho)$ and $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\ldots)$. We case-analyze on T_{ST}.

Subcase: $T_{ST} = \overline{S}$ or $T_{ST} = Owned$. By E-$\rightarrow_{owned}$, $\Sigma, l \rightarrow_{owned} S(\overline{l'}) \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(\overline{l'})]/\mu]\,\Sigma, ()$

Subcase: $T_{ST} = Shared$.

Case: $\rho(l) \notin \phi$. Then by E-$\rightarrow_{shared}$,
$\Sigma, l \rightarrow_{shared} S(\overline{l'}) \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(\overline{l'})]/\mu]\,\Sigma, ()$.

Case: $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S(\ldots)$. Then by E-$\rightarrow_{shared}$,
$\Sigma, l \rightarrow_{shared} S(\overline{l'}) \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(\overline{l'})]/\mu]\,\Sigma, ()$.

Case: e is stuck at a bad state transition. In that case, we have
$e = \mathbb{E}[l \rightarrow_{\text{Shared}} S(\overline{l'})]$ where $\mu(\rho(l)) = C^*\langle\overline{T^*}\rangle@S'(\ldots)$, $S \neq S'$, $\rho(l) \in \phi$, and $\Gamma; \Delta \vdash_s l : C\langle\overline{T'}\rangle.\text{Shared} \dashv \Delta'$. $C = C^*$ due to memory consistency.

Subcase: T_{ST} = Unowned. This case is impossible, because it contradicts the antecedent $T_{ST} \neq \text{Unowned of } T\text{-}\rightarrow_p$.

Case: T-assertStates. Because e is closed, $e = [l@\overline{S}]$. By rule E-assert, $\Sigma, \text{assert } l \text{ in } \overline{S} \rightarrow \Sigma, ()$.

Case: T-assertPermission. Because e is closed, $e = [l@T_{ST}]$. By rule E-assert, $\Sigma, \text{assert } l \text{ in } T_{ST} \rightarrow \Sigma, ()$.

Case: T-assertInVar. Because e is closed, $e = [l@T_{ST}]$. By rule E-assert, Σ, assert l in $T_{ST} \rightarrow \Sigma$, ().

Case: T-assertInVarAlready. Because e is closed, $e = [l@T_{ST}]$. By rule E-assert, Σ, assert l in $T_{ST} \rightarrow \Sigma$, ().

Case: T-IsIn-StaticOwnership. Because e is closed, $e = \text{if } l \text{ in}_{owned} S \text{ then } e_1 \text{ else } e_2$. By memory consistency, there exists S' such that $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S'(\ldots)$.

Subcase: $S' = S$. Then by E-IsIn-Dynamic-Match-Owned, $\Sigma, e \rightarrow \Sigma, e_1$.

Subcase: $S' \neq S$. By IsIn-Dynamic-Else, $\Sigma, e \rightarrow \Sigma, e_2$.

Case: T-isInDynamic. Because e is closed, $e = \text{if } l \text{ in}_{shared} S \text{ then } e_1 \text{ else } e_2$. By memory consistency, $l \in dom(\rho)$ and there exists S' such that $\mu(\rho(l)) = C\langle\overline{T'}\rangle@S'(\ldots)$.

By inversion, we have $l : C\langle\overline{T'}\rangle$.Shared.

Subcase: $S' = S$. Then if $\rho(l) \in \phi$ then we are stuck in a nested dynamic state check. Otherwise, by E-IsIn-Dynamic-Match-Shared, $\Sigma, e \rightarrow [\phi, \rho(l)/\phi]\,\Sigma, \boxed{e_1}_{\rho(l)}$.

Subcase: $S' \neq S$. By IsIn-Dynamic-Else, $\Sigma, e \rightarrow \Sigma, e_2$.

Case: T-IsIn-PermVar. Because e is closed, $e = \text{if } l \text{ in}_P p \text{ then } e_1 \text{ else } e_2$. By assumption Γ, Σ, Δ **ok**, so $\xi(p) = T_{ST}$ for some T_{ST}. Then Σ, if $l \text{ in}_P p$ then e_1 else $e_2 \rightarrow \Sigma$, if $l \text{ in}_P T_{ST}$ then e_1 else e_2.

Case: T-IsIn-Perm-Then. Because e is closed, $e = \text{if } l \text{ in}_p \text{ Perm then } e_1 \text{ else } e_2$. By inversion, $\Gamma \vdash P <:_* \text{Perm}$. As both P and Perm are permissions, not variables, we have $\cdot \vdash P <:_* \text{Perm}$, so by E-IsIn-Permission-Else $\Sigma, e \rightarrow \Sigma, e_1$.

Case: T-IsIn-Perm-Else. Because e is closed, $e = \text{if } l \text{ in}_p \text{ Perm then } e_1 \text{ else } e_2$. By inversion, $\Gamma \vdash \text{Perm} <:_* P$, and $P \neq \text{Perm}$. As both P and Perm are permissions, not variables, we have $\cdot \vdash \text{Perm} <:_* P$, so by E-IsIn-Permission-Else $\Sigma, e \rightarrow \Sigma, e_2$.

Case: T-IsIn-Unowned. Because e is closed, $e = \text{if } l \text{ in}_p \text{ Perm then } e_1 \text{ else } e_2$. In this case, by E-IsIn-Unowned $\Sigma, e \rightarrow e_2$.

Case: T-disown. Because e is closed, $e = \text{disown } l$. By rule *disown*, Σ, disown $l \rightarrow \Sigma$, ().

Case: T-pack. By *pack*, Σ, pack $\rightarrow \Sigma$, ().

Case: T-state-mutation-detection. Because e is closed, $e = \boxed{e'}_o$, where e' is also closed. If e' is a value v, then by E-Box-ϕ, $\Sigma, \boxed{\text{v}}_o \rightarrow [(\phi \setminus o)/\phi]\,\Sigma, v$. Otherwise, by the induction hypothesis, either $\Sigma, e' \rightarrow \Sigma', e''$, or e' is stuck with an appropriate evaluation context. In the former case, by E-box-ϕ-congr, $\Sigma, \boxed{e'}_o \rightarrow \Sigma', \boxed{e''}_o$. In the latter case, e is stuck with an appropriate evaluation context.

Case: T-reentrancy-detection. Because e is closed, $e = \boxed{e'}^o$, where e' is also closed. If e' is a value v, then by E-Box-ψ, $\Sigma, \boxed{\text{v}}^o \rightarrow [(\psi \setminus o)/\psi]\,\Sigma, v$. Otherwise, by the induction hypothesis, either $\Sigma, e' \rightarrow \Sigma', e''$, or e' is stuck with an appropriate evaluation context. In the former case, by E-box-ψ-congr, $\Sigma, \boxed{\text{e'}}^o \rightarrow \Sigma', \boxed{\text{e"}}^o$. In the latter case, e is stuck with an appropriate evaluation context. □

Theorem C.2 (Preservation). *If e is a closed expression, $\Gamma; \Delta \vdash_s e : T \dashv \Delta''$, Γ, Σ, Δ ok, and $\Sigma, e \rightarrow \Sigma', e'$ then for some Δ', $\Gamma'; \Delta' \vdash_s e' : T' \dashv \Delta'''$, $\Gamma', \Sigma', \Delta'$ ok, and $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$.*

Proof. Proof proceeds by induction on the dynamic semantics.

Case: E-lookup. $e = l$. We case-analyze on T.

Subcase: T = unit

By assumption, Γ, Σ, Δ ***ok***, and $\Gamma; \Delta \vdash_s l : T \dashv \Delta''$. By assumption and E-Lookup, $\Sigma, l \rightarrow \Sigma, \Sigma_\rho(l)$. The fact that $\Sigma_\rho(l) = ()$ follows directly from global consistency. Then by T-(), $\Gamma; \Delta \vdash_s () : \text{unit} \dashv \Delta$. Global consistency is immediate, because the contexts are unchanged, and $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Subcase: $T = C\langle\overline{T'}\rangle@T_{ST}$

By inversion, $\Delta = \Delta_0, l : T_0$, $T_0 \Rrightarrow T/T_2$, and $\Delta'' = \Delta_0, l : T_2$. By rule E-lookup, $e' = \Sigma_\rho(l)$. The fact that $\Sigma_\rho(l) = o$ for some o follows directly from global consistency. $l \neq o$ by construction and if o occurs in Δ_0, then we apply the strengthening lemma to generate a new proof of $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$ in which o does not occur. Thus, $\Delta' = \Delta'', o : T_2$ is a valid typing context. Then by Var, $\Gamma; \Delta'', o : T_2 \vdash_s o : T_2 \dashv \Delta'', o : T'$ for some T'. Now, Δ' is the same as Δ except that some instances of T_0 have been replaced with T_2. The required consistency is obtained from the Split Compatibility lemma (C.12). We have $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$, because the two contexts differ only on o, which is not relevant to the $<^l$ relation.

Subcase: $T = I\langle\overline{T'}\rangle@T_{ST}$ or $T = X@T_{ST}$

By memory consistency, this case is impossible.

Case: E-assign. $e = l := l'$. By assumption:

(1) $\Sigma, l := l' \rightarrow [\rho[l \mapsto o]/\rho]\,\Sigma, ()$.

(2) $\Gamma; \Delta^*, l : T_l, l' : T_{l'} \vdash_s l := l' : \mathsf{unit} \dashv \Delta^{**}, l : T^*, l' : T^{**}$.

By inversion:

(1) $T_{l'} \Rrightarrow T^*/T^{**}$.

(2) $\Gamma \vdash \mathit{disposable}(T_l)$.

Let $\Delta' = \Delta^{**}, l : T^*, l' : T^{**}$. By rule T-(), $\Gamma; \Delta' \vdash_s () : \mathsf{unit} \dashv \Delta'$. We obtain consistency as a corollary of the split compatibility lemma. Finally, $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ due to reflexivity of $<^l$.

Case: E-new. $e = \mathsf{new}\, C\langle\overline{T'}\rangle@S(\overline{l})$, because e is closed (any variables would be free, so they must not exist).

By assumption, $\Gamma; \Delta \vdash_s \mathsf{new}\ C\langle\overline{T'}\rangle@S(\overline{l}) : C\langle\overline{T'}\rangle@S \dashv \Delta''$; also, $e' = o$, and $o \notin \mathit{dom}(\mu)$. Let $\Delta' = \Delta'', o : C\langle\overline{T'}\rangle@S$. By *T-lookup*, $\Gamma; \Delta'', o : C\langle\overline{T'}\rangle@S \vdash_s o : C\langle\overline{T'}\rangle@S \dashv \Delta'', o : C\langle\overline{T'}\rangle@\mathit{Unowned}$. Since o is fresh and Γ, Σ, Δ ***ok***, there are no refer ences to o in the previous contexts, so all of the aliases are trivially consistent. We also have $\Gamma \vdash T <: \mathit{stateFields}(C\langle\overline{T'}\rangle, S)$, where $\overline{l : T} \in \Delta$, which implies the required field property for reference consistency. By the split compatibility lemma, we have Γ, Σ, Δ'***ok***. We have $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$, because the two contexts differ only on o, which is not relevant to the $<^l$ relation.

Case: E-let. $e = \mathsf{let}\ x : T_1 = v\ \mathsf{in}\ e_2$ By assumption:

(1) $\Sigma, \mathsf{let}\ x : T_1 = v\ \mathsf{in}\ e_2 \rightarrow [\rho[l \mapsto v]/\rho]\,\Sigma, [l/x]e$.

(2) Γ, Σ, Δ ok.

(3) $\Gamma; \Delta \vdash_s \mathsf{let}\ x : T_1 = v\ \mathsf{in}\ e_2 : T \dashv \Delta''$.

Subcase: v = o.

(1) By inversion:
   (a) $\Gamma; \Delta \vdash_s o : T_1 \dashv \Delta^*$
   (b) $\Gamma; \Delta^*, x : T_1 \vdash_s e_2 : T \dashv \Delta^{**}, x : T_1'$
   (c) $\Gamma \vdash \mathit{disposable}(T_1')$
   (d) $l \notin \mathit{dom}(\rho)$

(2) Let $\Delta' = \Delta^*, l : T_1$. By the substitution lemma (C.13) applied to 1b, $\Gamma; \Delta' \vdash_s [l/x]e_2 : T \dashv \Delta^{**}, l : T_1'$.

(3) By global consistency and 1a, T_1 is consistent with all other references in $\mathit{refTypes}(\Sigma, \Delta, o)$. Now, note that by global consistency, all references were previously compatible with T_1. Σ' now includes a reference to the same object with indirect reference l, which corresponds with $l : T_1 \in \Delta'$. The only rule that could

have been used in 1a is T-lookup, which split $T_1 \Rrightarrow T_1'/T_3$ and replaced $o : T_1 \in \Delta$ with $o : T_3 \in \Delta'$. By the split compatibility lemma (C.12), T_3 is compatible with all other aliases to o, and in particular with T_1'.

(4) $\Delta^{**}, l : T_1' <^{l}_{\Gamma,\Sigma'} \Delta^{**}, x : T_1'$, because $l \notin dom(\Delta^{**}, x : T_1')$.

Subcase: v = (). By inversion:

(1) $\Gamma; \Delta \vdash_s () : \text{unit} \dashv \Delta$
(2) $\Gamma; \Delta, x : \text{unit} \vdash_s e_2 : T \dashv \Delta^*, x : T_1'$
(3) $\Gamma \vdash \mathit{disposable}(\text{unit})$
(4) $l \notin dom(\rho)$

Let $\Delta' = \Delta^*, l : \text{unit}$. By the substitution lemma (C.13) $\Gamma; \Delta^*, l : \text{unit} \vdash_s [l/x]e_2 : T \dashv \Delta^{**}, l : T_1'$. Then the extensions to the contexts do not affect permissions, so they must be compatible, and Γ, Σ', Δ' ok. $\Delta^{**}, l : T_1' <^{l}_{\Gamma,\Sigma'} \Delta^{**}, x : T_1'$, because $l \notin dom(\Delta^{**}, x : T_1')$.

Case: E-letCongr. $e = \text{let } x : T_1 = e_1 \text{ in } e_2$.

(1) By assumption:
  (a) Γ, Σ, Δ ok
  (b) $\Gamma; \Delta \vdash_s \text{let } x : T_1 = e_1 \text{ in } e_2 : T \dashv \Delta''$
(2) By inversion:
  (a) $\Sigma, e_1 \to \Sigma^*, e_1'$.
  (b) $\Gamma; \Delta \vdash_s e_1 : T_1 \dashv \Delta^*$
  (c) $\Gamma; \Delta^*, x : T_1 \vdash_s e_2 : T_2 \dashv \Delta^{**}, x : T_1'$
  (d) $\Gamma \vdash \mathit{disposable}\left(T_1'\right)$
(3) By the induction hypothesis:
  (a) $\Gamma'^*; \Delta'^* \vdash_s e_1' : T_1 \dashv \Delta''^*$ for some Γ'^*, Δ'^*, and Δ''^*
  (b) $\Gamma'^*, \Sigma^*, \Delta'^*$ ok
  (c) $\Delta''^* <^{l}_{\Gamma,\Sigma'} \Delta^*$
(4) By C.5 with 3c and 2c, we have $\Gamma; \Delta''^*, x : T_1 \vdash_s e_2 : T_2 \dashv \Delta^{***}, x : T_1'$, with $\Delta^{***} <^{l}_{\Sigma^*} \Delta^{**}$.
(5) Let $\Delta' = \Delta'^*$ and let $\Gamma^{**} = \Gamma, \Gamma'^*$.
  Then, by rule Let with 3a, 4, and 2d, $\Gamma{*}{*}; \Delta' \vdash_s \text{let } x : T_1 = e_1' \text{ in } e_2 : T \dashv \Delta^{***}$, where $\Delta^{***} <^{l}_{\Sigma'} \Delta''$.
(6) By C.15, $\Gamma^{**}, \Sigma', \Delta'$ ok.

Case: E-Inv. $e = l_1.m\langle\overline{M}\rangle(\overline{l_2})$, because e is closed.

(1) By assumption, and because e is closed:
  (a) $\Sigma, l_1.m\langle\overline{M}\rangle(\overline{l_2}) \to [\psi, \rho(l_1)/\psi]\,\Sigma'', \boxed{\overline{[l_2'/x]}[l_1'/\text{this}]e}^{\rho(l_1)}$
  (b) Γ, Σ, Δ ok
  (c) $\Gamma; \Delta_0, l_1 : C\langle\overline{T}\rangle@T_{ST l1}, \overline{l_2 : T_{l2}} \vdash_s l_1.m\langle\overline{M}\rangle(\overline{l_2}) : T \dashv \Delta_0, l_1 : T_{l1}', \overline{l_2 : T_{l2}'}$
(2) By inversion:
  (a) $l_1' \notin dom(\rho)$
  (b) $\overline{l_2' \notin dom(\rho)}$
  (c) $params(C) = \overline{T_D}$
  (d) $\Sigma'' = \Sigma[l_1' \mapsto \rho(l_1)][\overline{l_2' \mapsto \rho(l_2)}]$
  (e) $\xi' = \xi, \overline{PermVar(T_D) \mapsto Perm(T)}, \overline{PermVar(T_M) \mapsto Perm(M)}$
  (f) $\Sigma''' = [\xi'/\xi]\,[\psi, \rho(l_1)/\psi]\,\Sigma'$
  (g) $\mu(\rho(l_1)) = C\langle\overline{T}\rangle@S(\ldots)$
  (h) $\rho(l_1) \notin \psi$
  (i) $tdef(C, m) = m\langle\overline{T_M}\rangle(\overline{T_x \gg T_{xST}\ x})\ T_{this} \gg T'_{this}\ e'$

(j) $\Gamma \vdash C\langle\overline{T}\rangle@T_{STl1} <: C\langle\overline{T}\rangle@T_{this}$
(k) $\overline{\Gamma \vdash T_{l2} <: C_x@T_x}$
(l) $T'_{l1} = funcArg(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this})$
(m) $\overline{T'_{l2} = funcArg(T_{l2}, T_x, C_x@T_{xST})}$

(3) We assume that the transaction is well-typed in its contract:
$T\ m\langle\overline{M}\rangle(\overline{C_x@T_x \gg T_{xST}\ x})T_{this} \gg T'_{this}\ e$ **ok in** C. As a result, we additionally have (by inversion):
(a) $\overline{T_D}, \overline{T_G}; this : C\langle\overline{T}\rangle@T_{this},\ \overline{x : C_x@T_x} \vdash_{s_1} e : T \dashv this : C\langle\overline{T}\rangle@T'_{this}, \overline{x : C_x@T_{xST}}$
Then by the substitution lemma for interfaces (C.9), we also have
(a) $\overline{T_D}, \overline{T_G}; this : C\langle\overline{T}\rangle@T_{this},\ \overline{x : C'\langle\overline{T'}\rangle@T_x} \vdash_{s_1} e : T \dashv this : C\langle\overline{T}\rangle@T'_{this}, \overline{x : C'\langle\overline{T'}\rangle@T_{xST}}$
where $l_2 : C'\langle\overline{T'}\rangle@T'_{ST}$, by global consistency.

(4) Let $\Gamma' = \Gamma, \overline{T_D}, \overline{T_M}$. By the substitution lemma (C.13) on 3a, we have:
$\Gamma'; l'_1 : C\langle\overline{T}\rangle@T_{this},\ \overline{l'_2 : C'\langle\overline{T'}\rangle@T_x} \vdash_{s_1} [l'_2/x][l'_1/\text{this}]e : T \dashv l'_1 : C\langle\overline{T}\rangle@T'_{this}, \overline{l'_2 : C'\langle\overline{T'}\rangle@T_{xST}}$

(5) *funcArgResidual* is defined in AJ:35. Let:

$$T_{l1R} = funcArgResidual(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this})$$
$$\overline{T_{l2R}} = \overline{funcArgResidual\,(T_{l2}, T_x, C_x@T_{xST})}$$
$$\Delta' = \Delta, l_1 : T_{l1R},\ \overline{l_2 : T_{l2R}}, l'_1 : C\langle\overline{T}\rangle@T_{this}, \overline{l'_2 : C'\langle\overline{T'}\rangle@T'_{l2}}.$$

Note that l_1 and l_2 do not occur free in $[\overline{l'_2/x}][l'_1/\text{this}]e$, because otherwise (3a) would not have been the case. Then we have (by weakening 4): $\Gamma'; \Delta' \vdash_s [\overline{l'_2/x}][l'_1/\text{this}]e : T \dashv \Delta, l_1 : T_{l1R}, \overline{l_2 : T_{l2R}}, l'_1 : C\langle\overline{T}\rangle@T'_{this}, \overline{l'_2 : C'\langle\overline{T'}\rangle@T_{xST}}$

(6) By rule Reentrancy-detection:
$\Gamma'; \Delta' \vdash_s \boxed{[\overline{l'_2/x}][l'_1/\text{this}]e}^{\rho(l)} : T \dashv \Delta, l_1 : T_{l1R}, \overline{l_2 : T_{l2R}}, l'_1 : C\langle\overline{T}\rangle@T'_{this}, \overline{l'_2 : C'\langle\overline{T'}\rangle@T_{xST}}$
which corresponds to the evaluation step in 1a. This also gives us that every indirect reference has a contract type, as required by global consistency.

(7) Consider:

$$T_{l1R} = funcArgResidual(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this})$$
$$T'_{l1} = funcArg(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this})$$

If $T'_{l1} \neq C\langle\overline{T}\rangle@T'_{this}$, then there are two possibilities, both with $C\langle\overline{T}\rangle@T_{this} =$ Unowned. If $C\langle\overline{T}\rangle@T_{STl1} = T_C$@Shared, then $T_{l1R} = T_C$@Shared; otherwise, *maybeOwned* (T_{l1R}). In both cases, $T_{l1R} \approx T'_{l1}$ and $\Gamma \vdash T_{l1R} <: T'_{l1}$. The same argument holds for l_2 and its type. Therefore:
$\Delta, l_1 : T_{l1R}, \overline{l_2 : T_{l2R}}, l'_1 : C\langle\overline{T}\rangle@T'_{this}, \overline{l'_2 : C'\langle\overline{T'}\rangle@T_{xST}} <:^{l}_{\Gamma,\Sigma'} \Delta_0, l_1 : T'_{l1}, \overline{l_2 : T'_{l2}}$

(8) By assumption of Γ, Σ, Δ ***ok***, ξ contains mappings for each $p \in PermVar\,(\Gamma)$. Note that ξ' additionally contains mappings for each $\overline{T_G}$ and $\overline{T_D}$, so $PermVar\,(\Gamma') \subset \{p \mid \xi(p) = T_{ST}\}$, as required by global consistency. Finally, to show $\Gamma', \Sigma', \Delta'$ ok, we need to show that the new types for l_1 and $\overline{l_2}$ are compatible with the aliases in Δ'.

First consider T_{l1R} and $C\langle\overline{T}\rangle@T_{this}$, which alias the object originally referenced with type $C\langle\overline{T}\rangle@T_{STl1}$. By assumption (1c and 1b), $C\langle\overline{T}\rangle@T_{STl1}$ is compatible with all existing aliases in Σ. Note that $T_{l1R} = funcArgResidual(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this})$.

Consider the cases for T_{l1R}:

Case: FuncArg-owned-unowned. Previously, $l_1 : C\langle\overline{T}\rangle@T_{STl1}$ was in Δ, and Γ', Σ, Δ **ok**. Now, Δ' includes both $C\langle\overline{T}\rangle@T_{this}$ and $C\langle\overline{T}\rangle@T_{STl1}$. But $T_{this} =$ Unowned, which is compatible with all other references.

Case: FuncArg-shared-unowned. Previously, $l_1 : C\langle\overline{T}\rangle@Shared$ was in Δ, and Γ', Σ, Δ **ok**. Now, Δ' includes both $C\langle\overline{T}\rangle@T_{this}$ and $C\langle\overline{T}\rangle$.Shared. But $T_{this} =$ Unowned, which is compatible with Shared.

Case: FuncArg-other. Previously, $l_1 : C\langle\overline{T}\rangle@T_{STl1}$ was in Δ, and Γ', Σ, Δ ok. Now, Δ' includes both $C\langle\overline{T}\rangle@T_{this}$ and $C\langle\overline{T}\rangle$.Unowned. But Unowned is compatible with all other references.

The corresponding argument applies to l_2'.

Case: E-Inv-Private. $e = l_1.m\langle\overline{M}\rangle(\overline{l_2})$, because e is closed.

This case is similar to the E-Inv case, except that the fields are treated in a manner analogous to arguments: the field states are part of the initial context; they are transformed via $funcArg$; and the resulting types are in the output context.

Case: E-IsIn-Dynamic-Match-Owned. $e =$ if x in$_{owned}$ T_{ST} then e_1 else e_2, because e is closed.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_s$ if l is in$_{owned}$ S then e_1 else $e_2 : T_1 \dashv \Delta''$
  (c) Σ, if l is in$_{owned}$ S then e_1 else $e_2 \to \Sigma, e_1$
(2) By inversion:
  (a) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$
  (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@S \vdash_s e_1 : T_1 \dashv \Delta^*$
  (c) $S \in states(C\langle\overline{T}\rangle)$
  (d) $\overline{S_x} = possibleStates_\Gamma\left(C\langle\overline{T}\rangle@T_{ST}\right)$
  (e) $\Gamma \vdash T_{ST} <:_* Owned$
  (f) $\Gamma; \Delta_0, x : C\langle\overline{T}\rangle.(\overline{S_x} \setminus S) \vdash_s e_2 : T_1 \dashv \Delta^{**}$
  (g) $\Delta'' = merge(\Delta^*, \Delta^{**})$
(3) Let $\Delta' = \Delta_0, l : C\langle\overline{T}\rangle@S$. By 2b, $\Gamma; \Delta' \vdash_s e_1 : T_1 \dashv \Delta^*$.
(4) The difference between Δ and Δ' is that in Δ', the type of l is $C\langle\overline{T}\rangle@S$. To show that Γ, Σ', Δ' **ok**, we need to show that $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$. But this is given by (2a).
(5) By the merge subtyping lemma C.19, if $l : T \in merge(\Delta^*, \Delta^{**})$, then $l : T' \in \Delta^*$ with $\Gamma \vdash T' <: T$ and $T' \approx T$. Thus, $\Delta^* <:^l_{\Gamma;\Sigma} \Delta''$.

Case: E-IsIn-Dynamic-Match-Shared. $e =$ if l is in$_{shared}$ $\overline{S}$ then e_1 else e_2

(1) By assumption, and because e is closed:
  (a) $\Gamma, \Sigma, \Delta_0, l : C\langle\overline{T}\rangle@Shared$ **ok**
  (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Shared \vdash_s$ if l is in$_{shared}$ S then e_1 else $e_2 : T_1 \dashv \Delta''$
  (c) Σ, if l is in$_{shared}$ S then e_1 else $e_2 \to [\phi, \rho(l)/\phi]\,\Sigma, \boxed{e_1}_{\rho(l)}$
(2) By inversion:
  (a) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@S \vdash_s e_1 : T_1 \dashv \Delta^*, l : C\langle\overline{T}\rangle@T_{ST}$
  (b) $\Gamma \vdash bound\,(T_{ST}) \neq$ Unowned
  (c) $S \in stateNames\,(C)$
  (d) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Shared \vdash_s e_2 : T_1 \dashv \Delta^{**}, l : C\langle\overline{T}\rangle@Shared$
  (e) $\Delta'' = merge(\Delta^*, \Delta^{**}), l : C\langle\overline{T}\rangle@Shared$
  (f) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$
  (g) $\rho(l) \notin \phi$

(3) Let $\Delta' = \Delta_0, l : C\langle\overline{T}\rangle@S$. By State-mutation-detection and 2a, $\Gamma; \Delta' \vdash_s \boxed{e_1}_{\rho(l)} : T_1 \dashv \Delta'''$.
(4) The difference between Δ and Δ' is that in Δ', the type of l is $C\langle\overline{T}\rangle@S$. By (2f), we know that $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$. However, there may be other aliases to $\rho(l)$ that have Shared permission. Since $\rho(l)$ is in the ϕ context of Σ', any other references to $\rho(l)$ must be compatible with $C\langle\overline{T}\rangle@Shared$, so we have Γ, Σ', Δ' **ok** via *StateLockCompatible*.
(5) By the merge subtyping lemma C.19, if $l : T \in merge(\Delta^*, \Delta^{**})$, then $l : T' \in \Delta^*$ with $\Gamma \vdash T' <: T$. Thus, $\Delta^* <^l_{\Gamma;\Sigma} \Delta''$.

Case: E–IsIn-Dynamic-Else. $e = \text{if } l \text{ is in}_p\ \overline{S} \text{ then } e_1 \text{ else } e_2$
(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Sigma, \text{if } l \text{ is in}_p\ S \text{ then } e_1 \text{ else } e_2 \rightarrow \Sigma, e_2$
  (c) $\Gamma; \Delta \vdash_s \text{if } l \text{ is in}_p\ S \text{ then } e_1 \text{ else } e_2 : T_1 \dashv \Delta''$
(2) By inversion:
  (1) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots)$
  (2) $S' \notin \overline{S}$
(3) By inversion, either:
  (a) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Shared \vdash_s e_2 : T_1 \dashv \Delta^*$; or:
  (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle.\overline{S_x} \setminus \overline{S} \vdash_s e_2 : T_1 \dashv \Delta^{**}$
(4) If we are in case (3a), then let $\Delta' = \Delta$. Then by 3a, $\Gamma; \Delta' \vdash_s e_2 : T_1 \dashv \Delta^*$. By assumption, Γ, Σ, Δ' ok. By the merge subtyping lemma C.19, if $l : T \in merge(\Delta^*, \Delta^{**})$, then $l : T' \in \Delta^*$ with $\Gamma \vdash T' <: T$. Thus, $\Delta^* <^l_{\Gamma;\Sigma} \Delta''$.
(5) Otherwise, let $\Delta' = \Delta_0, l : C\langle\overline{T}\rangle.\overline{S_x} \setminus \overline{S}$. Then by 3b, $\Gamma; \Delta' \vdash_s e_2 : T_1 \dashv \Delta^{**}$. By inversion, we had $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_s e_2 : T_1 \dashv \Delta^{**}$. As a result, there are no other owning references to the object referenced by l, and the referenced object is in state S' by (2a). Since $S' \notin \overline{S}$, $C\langle\overline{T}\rangle.\overline{S_x} \setminus \overline{S}$ is a consistent type for the reference, and Γ, Σ, Δ' ok. By the merge subtyping lemma C.19, if $l : T \in merge(\Delta^*, \Delta^{**})$, then $l : T' \in \Delta^{**}$ with $\Gamma \vdash T' <: T$. Thus, $\Delta^{**} <^l_{\Gamma;\Sigma} \Delta''$.

Case: E-IsIn-PermVar
(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta \vdash_s \text{if } l \text{ is in}_{Perm}\ p \text{ then } e_1 \text{ else } e_2 : T_1 \dashv \Delta''$
  (c) $\Sigma, \text{if } l \text{ is in}_{Perm}\ p \text{ then } e_1 \text{ else } e_2 \rightarrow \Sigma, \text{if } l \text{ is in}_{Perm}\ T_{ST} \text{ then } e_1 \text{ else } e_2$
(2) By inversion:
  (a) $\xi(p) = T_{ST}$
  (b) $\Gamma; \Delta, l : T_C@p \vdash_s e_1 : T_1 \dashv \Delta'$
  (c) $\Gamma; \Delta, l : T_C@T'_{ST} \vdash_s e_2 : T_1 \dashv \Delta''$
  (d) $\Delta_f = merge(\Delta', \Delta'')$
  (e) $\text{Perm} = toPermission(T'_{ST})$
(3) To perform substitution for type parameters, we must have proved $subsOk_\Gamma(T, T_G)$, so we must have $\Gamma \vdash T_{ST} <:_* p$. Then by 2b and the permission variable substitution lemma C.10, we have $\Gamma; \Delta, l : T_C@T_{ST} \vdash_s e_1 : T_1 \dashv \Delta'$.
(4) We proceed by case analysis on T_{ST}.

Subcase: $T_{ST} = \overline{S}$

If $P =$ Unowned, then $T'_{ST} =$ Unowned, and by 2c we can apply T-IsIn-Unowned to show $\Gamma; \Delta, l : T_C@\text{Unowned} \vdash_s$ if l is $\text{in}_{Unowned}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.
If $P =$ Shared, then $T'_{ST} =$ Shared, and by 2c we can apply T-IsIn-Dynamic to show $\Gamma; \Delta, l : T_C@\mathit{Shared} \vdash_s$ if l is $\text{in}_{shared}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.
If $P =$ Owned, then $\Gamma \vdash T'_{ST} <:$ Owned, so $\mathit{maybeOwned}(T_C@T'_{ST})$, and $\Gamma \vdash \overline{S_x} <:_* T'_{ST}$, where $\overline{S_x} = \mathit{possibleStates}_\Gamma(T_C@T_{ST})$. Then by the subtype substitution lemma C.4 and by 2c we have $\Gamma; \Delta, l : T_C@(\overline{S_x} \setminus \overline{S}) \vdash_s e_2 : T_1 \dashv \Delta''$. Now we can apply T-IsIn-StaticOwnership to get $\Gamma; \Delta, l : T_C@T'_{ST} \vdash_s$ if l is $\text{in}_{owned}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.

Subcase: $T_{ST} = P$

If $\Gamma \vdash \text{Perm} <:_* P$, then by IsIn-Permission-Then,
$\Gamma; \Delta, l : T_C@T_{ST} \vdash_s$ if l is $\text{in}_{\text{Perm}}\ P$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.
Otherwise, $\Gamma \vdash \text{Perm} \not<:_* P$, so by 2c and IsIn-Permission-Else, $\Gamma; \Delta, l : T_C@T_{ST} \vdash_s$ if l is in_P Perm then e_1 else $e_2 : T_1 \dashv \Delta'_f$.

Subcase: $T_{ST} = q$. This case is impossible, because ξ only maps to nonvariable states and permissions.

(5) In all cases, global consistency is maintained, because the environment does not change, $\Delta'_f <^l_{\Gamma,\Sigma'} \Delta'$ by reflexivity.

Case: E-IsIn-Permission-Then. By assumption, and because e is closed:

(1) Γ, Σ, Δ **ok**
(2) $\Gamma; \Delta \vdash_s$ if l is in_P Perm then e_1 else $e_2 : T_1 \dashv \Delta''$
(3) $\Sigma,$ if l is in_P Perm then e_1 else $e_2 \to \Sigma, e_1$

By inversion:

(1) Perm $\in$ {Owned, Unowned, Shared}
(2) $\cdot \vdash P <:_*$ Perm

To prove that e is well typed, we must have used either IsIn-Permission-Then or IsIn-Permission-Else. However, we know that $\cdot \vdash P <:_*$ Perm, so we must have used IsIn-Permission-Else. Then by inversion of IsIn-Permission-Then, we have $\Gamma; \Delta_0, x : T_C@T_{ST} \vdash_s e_1 : T_1 \dashv \Delta'''$.
Let $\Delta' = \Delta_0, x : T_C@T_{ST}$. Global consistency is maintained, because the environment has not changed, and $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-IsIn-Permission-Else. By assumption, and because e is closed:

(1) Γ, Σ, Δ **ok**
(2) $\Gamma; \Delta \vdash_s$ if l is in_P Perm then e_1 else $e_2 : T_1 \dashv \Delta''$
(3) $\Sigma,$ if l is in_P Perm then e_1 else $e_2 \to \Sigma, e_2$

By inversion:

(1) Perm $\in$ {Owned, Unowned, Shared}
(2) $\cdot \vdash \text{Perm} <:_* P$
(3) $P \neq$ Perm

To prove that e is well-typed, we must have used either IsIn-Permission-Then or IsIn-Permission-Else. However, we know that $\cdot \vdash \text{Perm} <:_* P$ and $P \neq$ Perm, so we must have used IsIn-Permission-Else. Then by inversion of IsIn-Permission-Else, we have $\Gamma; \Delta, x : T_C@T_{ST} \vdash_s e_2 : T_1 \dashv \Delta'$. Global consistency is maintained, because the environment has not changed, and $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-IsIn-Unowned. By assumption, and because e is closed:

(1) Γ, Σ, Δ **ok**
(2) $\Gamma; \Delta \vdash_s$ if l is $\text{in}_{\text{Unowned}}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta''$
(3) Σ, if l is $\text{in}_{\text{Unowned}}\ \overline{S}$ then e_1 else $e_2 \to \Sigma, e_2$

By inversion:

(1) $\Gamma; \Delta, x : T_C@T_{ST} \vdash_s e_2 : T_1 \dashv \Delta''$

e_2 is well typed by 1. Global consistency is maintained, because the environment has not changed, and $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-Box-ϕ. $e = \boxed{v}_o$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta \vdash_s \boxed{v}_o : T \dashv \Delta''$
  (c) $\Sigma, \boxed{\text{v}}_o \to [(\phi \setminus o)/\phi]\,\Sigma, v$
(2) By inversion:
  (a) $\Gamma; \Delta \vdash_s v : T \dashv \Delta''$
(3) Note that $\boxed{e}_o$ can only arise in the context of a shared-mode dynamic state test. Therefore, Δ must be of the form $\Delta_0, l : C\langle\overline{T}\rangle@Shared$ and Δ'' must be of the form $\Delta_0'', l : C\langle\overline{T}\rangle@Shared$.
(4) Since v is a value, either $v = ()$ or there exists o' such that $v = o'$. If $v = ()$, then let $\Delta' = \cdot$. By T-(), $\Gamma; \cdot \vdash_{()}$ unit $: \cdot \dashv \cdot$. Otherwise, $v = o'$ and by Var, there exists $o' : T_1 \in \Delta$ with $T_1 \Rrightarrow T_2/T_3$. In that case, let $\Delta' = \Delta, o' : T_1$. The proof proceeds as in the E-lookup rule: by Var, there exists $\Delta''' = o' : T_3$ such $\Gamma; \Delta' \vdash_s o' : T \dashv \Delta'''$.
(5) Δ''' differs from Δ'' only on bindings for o', which is not relevant to the $<^l$ relation, so $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.
(6) Γ, Σ, Δ' ok by the split compatibility lemma.

Case: E-Box-ϕ-congr. $e = \boxed{e}_o$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta \vdash_s \boxed{e}_o : T \dashv \Delta''$
  (c) $\Sigma, \boxed{\text{e}}_o \to \Sigma', \boxed{\text{e'}}_o$
(2) By inversion:
  (a) $\Sigma, e \to \Sigma', e'$
  (b) $\Gamma; \Delta \vdash_s e : T \dashv \Delta''$
(3) Let $\Delta' = \Delta$. By 1a, Γ, Σ, Δ' **ok**. Note that $\Delta''' = \Delta''$. $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity. By State-mutation-detection, $\Gamma; \Delta' \vdash_s \boxed{e'}_o : T \dashv \Delta''$.

Case: E-Box-ψ. $e = \boxed{v}^o$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta \vdash_s \boxed{v}^o : T \dashv \Delta''$
  (c) $\Sigma, \boxed{\text{v}}^o \to [(\psi \setminus o)/\psi]\,\Sigma, v$
(2) By inversion:
  (a) $\Gamma; \Delta \vdash_s v : T \dashv \Delta''$
(3) Let $\Delta' = \Delta$. By 2a, $\Gamma; \Delta' \vdash_s v : T \dashv \Delta''$. $\Sigma' = [(\psi \setminus o)/\psi]\,\Sigma$. Note that the definition of consistency does not depend on Σ_ψ. With 1a, we conclude that Γ, Σ', Δ' ok. Note that $\Delta''' = \Delta''$. $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-Box-ψ-congr. $e = \boxed{e}^o$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Gamma; \Delta \vdash_s \boxed{e}^o : T \dashv \Delta''$
  (c) $\Sigma, \boxed{\text{e}}^o \rightarrow \Sigma', \boxed{\text{e'}}^o$
(2) By inversion:
  (a) $\Gamma; \Delta \vdash_s e : T \dashv \Delta''$
  (b) $\Sigma, e \rightarrow \Sigma', e'$
(3) Let $\Delta' = \Delta$. By 1a, Γ, Σ, Δ'**ok**. Note that $\Delta''' = \Delta''$. $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity. By Reentrancy-detection, $\Gamma; \Delta' \vdash_s \boxed{e'}^o : T \dashv \Delta''$.

Case: E-State-Transition-Static-Ownership. $e = l \rightarrow_{owned} S(\overline{l'})$

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Sigma, l \rightarrow_{owned} S(\overline{l'}) \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()$
  (c) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_l l \rightarrow_{owned} S(\overline{x}) : \text{unit} \dashv \Delta^*, l : C\langle\overline{T}\rangle@S$
(2) By inversion:
  (a) $\Gamma \vdash T_{ST} <:_* \text{Owned}$
  (b) $\Gamma; \Delta_0 \vdash_l \overline{x} : \overline{T} \dashv \Delta^*$
  (c) $\Gamma \vdash T <: type(stateFields(C\langle\overline{T}\rangle, S'))$
  (d) $unionFields(C\langle\overline{T}\rangle, T_{ST}) = \overline{T_{fl}\ f_l}$
  (e) $fieldTypes_l(\Delta^*; \overline{T_{fl}\ f_l}) = \overline{T'_{fl}}$
  (f) $\overline{\Gamma \vdash disposable(T'_{fl})}$
(3) Let $\Delta' = \Delta, l : C\langle\overline{T}\rangle@S$. By T-(), $\Gamma; \Delta \vdash_l () : \text{unit} \dashv \Delta$. To show that Γ, Σ', Δ' **ok**, it suffices to show that any $T \in refTypes(\Sigma', \Delta', \rho(l))$ that specifies state specifies type $C\langle\overline{T}\rangle@S'$. But note that by 1c, l is in the original typing context with an owning type. Since Γ, Σ, Δ **ok**, and $C\langle\overline{T}\rangle@T_{ST} \in refTypes(\Sigma, \Delta, \rho(l))$, the only owning alias to the object referenced by l is l itself. Replacing $l : C\langle\overline{T}\rangle@T_{ST}$ in Δ with $l : C\langle\overline{T}\rangle@S$ replaces the type of the only owning alias with $C\langle\overline{T}\rangle@S$, which is consistent with $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\overline{l})$. $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-State-Transition-Shared. $e = l \rightarrow_{shared} S(\overline{l'})$

(a) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Sigma, l \rightarrow_{shared} S(\overline{l'}) \rightarrow [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()$
(2) Now, assume typing rule $\rightarrow_{shared}$ applied, since if $\rightarrow_{owned}$ applied, then the argument for case E-State-Transition-Static-Ownership (above) applies. Then:
  (a) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_l l \rightarrow_{shared} S(\overline{x}) : \text{unit} \dashv \Delta^*, l : C\langle\overline{T}\rangle@S$
(3) By inversion:
  (a) $\Gamma \vdash T_{ST} <:_* \text{Shared}$. By 2, we assume therefore $T_{ST} = \text{Shared}$.
  (b) $\Gamma; \Delta_0 \vdash_l \overline{x} : \overline{T} \dashv \Delta^*$
  (c) $\Gamma \vdash T <: type(stateFields(C\langle\overline{T}\rangle, S'))$
  (d) $unionFields(C\langle\overline{T}\rangle, T_{ST}) = \overline{T_{fl}\ f_l}$
  (e) $fieldTypes_l(\Delta^*; \overline{T_{fl}\ f_l}) = \overline{T'_{fl}}$
  (f) $\overline{\Gamma \vdash disposable(T'_{fl})}$
  (g) $\rho(l) \notin \phi \vee \mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$

(4) There are two subcases.

Subcase: $\rho(l) \notin \phi$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta \vdash_l () : \mathsf{unit} \dashv \Delta$. Now, all existing aliases to the object referenced by $\rho(l)$ were compatible with the previous reference, which was of type $C\langle\overline{T}\rangle@Shared$. As a result, none of those references restricted the state of the object, and the new state (in Σ') is consistent with Δ.

Subcase: $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\ldots)$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta \vdash_l () : \mathsf{unit} \dashv \Delta$. All references to the object referenced by $\rho(l)$ have the same type in Σ' as they did in Σ, because neither the contract nor the state of the object have changed, and we have Γ, Σ', Δ' **ok**.

(5) In both cases, $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-Field. $e = l.f_i$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Sigma, l.f_i \to \Sigma, o_i$

(2) By inversion:
  (a) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\overline{s})$

(3) Now, there are two subcases, because there are two possible type judgments for e.

Subcase: this-field-def
  (a) By assumption: $\Gamma; \Delta_0, l : T \vdash_l l.f : T_2 \dashv \Delta_0, l : T, l.f : T_3$
  (b) By inversion:
    (i) $l.f \notin Dom(\Delta)$
    (ii) $T_1\ f \in intersectFields(T)$
    (iii) $T_1 \Rrightarrow T_2/T_3$
  (c) Let $\Delta' = \Delta_0, l : T, l.f : T_3, o_i : T_2$. Then by Var, $\Gamma; \Delta' \vdash_s o_i : T_2 \dashv \Delta'''$ for some Δ'''. Γ, Σ, Δ' **ok**, because T_2 is a consistent permission for o_i per the split compatibility lemma (as in the E-lookup case). Δ''' agrees with Δ'' on all l, so $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Subcase: this-field-ctxt
  (a) By assumption: $\Gamma; \Delta_0, l : T, l.f : T_1 \vdash_l l.f : T_2 \dashv \Delta_0, l : T, l.f : T_3$
  (b) By inversion: $T_1 \Rrightarrow T_2/T_3$

  Let $\Delta' = \Delta_0, l : T, l.f : T_3, o_i : T_2$. Then by Var, $\Gamma; \Delta' \vdash_s o_i : T_3 \dashv \Delta'''$ for some Δ'''. Γ, Σ, Δ' **ok**, because T_2 is a consistent permission for o_i per the split compatibility lemma. Δ''' agrees with Δ'' on all l, so $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-FieldUpdate. $e = l.f_i := l'$.

(1) By assumption, and because e is closed:
  (a) Γ, Σ, Δ **ok**
  (b) $\Sigma, l.f_i := l' \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(o_1, o_2, \ldots, o_{i-1}, \rho(l'), o_{i+1}, \ldots, o_{|\overline{l.f}|})]/\mu]\,\Sigma, ()$
  (c) $\Gamma; \Delta \vdash_l l.f_i := l' : \mathsf{unit} \dashv \Delta^{**}, l.f_i : T_C@T_{ST}$

(2) By inversion:
  (a) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S(\overline{o})$
  (b) $fields(C\langle\overline{T}\rangle@S) = \overline{T\ f}$
  (c) $\Gamma; \Delta \vdash_l l.f_i : T_C@T_{ST} \dashv \Delta^*$
  (d) $\Gamma; \Delta^* \vdash_l l.f_i : T_C@T'_{ST} \dashv \Delta^{**}$
  (e) $\Gamma \vdash disposable\,(T_C@T_{ST})$

(3) Let $\Delta' = \Delta^*, l.f_i : T_C@T_{ST}$. By T-(), $\Gamma; \Delta' \vdash_l () : \mathsf{unit} \dashv \Delta'$.

(4) Note that $\Sigma' = [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(o_1, o_2, \ldots, o_{i-1}, \rho(l'), o_{i+1}, \ldots, o_{|l|})]/\mu]\,\Sigma$. By the same argument used in the proof of preservation for the *E-lookup* case, Γ, Σ, Δ^* **ok** and

likewise $\Gamma, \Sigma, \Delta^{**}$ **ok**. To show Γ, Σ', Δ' **ok**, we note that the only change relative to Σ and Δ^{**} is regarding the type of $l.f_i$. $\rho(l)$ has the same number of fields in Σ' as in Σ. Although $\rho(l)$ may now have an additional reference to $\rho(l')$ that did not exist before, this reference is compatible with all of the other references in $refTypes(\Sigma', \Delta^*, \rho(l'))$, because if the new reference is owned, this is only because $T_C@T_{ST}$ was owned, which was previously accounted for in $refTypes(\Sigma', \Delta^*, \rho(l'))$, and that ownership has been removed in Δ^{**}.

(5) Δ''' agrees with Δ'' on all l, so $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-Assert. $e = [x@T_{ST}]$.

(1) By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) Σ, assert l in $T_{ST} \rightarrow \Sigma, ()$

(2) There are two subcases:

Subcase: $T_{ST} = \overline{S}$**.** By assumption, $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle.\overline{S} \vdash_s [l@\overline{S'}] : \text{unit} \dashv \Delta_0, l : C\langle\overline{T}\rangle.\overline{S}$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta' \vdash_l () : \text{unit} \dashv \Delta'$. Since $\Sigma' = \Sigma$, $\Delta' = \Delta$, and Γ, Σ, Δ **ok**, we have Γ, Σ', Δ' **ok**.

Subcase: $T_{ST} \neq \overline{S}$**.** By assumption, $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_s [l@T_{ST}] : \text{unit} \dashv \Delta_0, l : C\langle\overline{T}\rangle@T_{ST}$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta' \vdash_l () : \text{unit} \dashv \Delta'$. Since $\Sigma' = \Sigma$, $\Delta' = \Delta$, and Γ, Σ, Δ **ok**, we have Γ, Σ', Δ' **ok**.

Case: E-Disown. $e = \text{disown } l$.

(1) By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, \text{disown } l \rightarrow \Sigma, l$

(2) There are two subcases:

Subcase: $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle.\overline{S} \vdash_s$ **disown** $l : \text{unit} \dashv \Delta_0, l : T'$**.** By inversion, $C\langle\overline{T}\rangle.\overline{S} \Rrightarrow T/T'$. Let $\Delta' = \Delta''$. By T-(), $\Gamma; \Delta' \vdash_s () : \text{unit} \dashv \Delta'$. Although the split compatibility lemma does not precisely apply here, an analogous argument does: any other alias to the object referenced by l was previously compatible with $C\langle\overline{T}\rangle.\overline{S}$, so we can see by case analysis of the definitions of compatibility and splitting that such aliases are also compatible with T'.

Subcase: $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Owned \vdash_s$ **disown** $l : \text{unit} \dashv \Delta_0, l : T'$**.** By inversion, $C\langle\overline{T}\rangle@Owned \Rrightarrow T/T'$. By T-(), $\Gamma; \Delta' \vdash_l () : \text{unit} \dashv \Delta'$. Although the split compatibility lemma does not precisely apply here, an analogous argument does: any other alias to the object referenced by l was previously compatible with $C\langle\overline{T}\rangle@Owned$, so we can see by case analysis of the definitions of compatibility and splitting that such aliases are also compatible with T'.

(3) In both subcases, $\Delta''' = \Delta''$, so $\Delta''' <^l_{\Gamma,\Sigma'} \Delta''$ by $<^l$-reflexivity.

Case: E-Pack. $e = \text{pack}$.

(1) By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, \text{pack } s \rightarrow \Sigma, ()$
   (c) $\Gamma; \Delta_0, l : T, \overline{l.f : T_f} \vdash_l \text{pack} : \text{unit} \dashv \Delta, l : T$. *(Note that $\overline{l.f : T_f}$ can be any subset of the declared fields, including the empty subset.)*

(2) By inversion:
   (a) $l.f \notin dom(\Delta_0)$
   (b) $contractFields(T) = \overline{T_{decl}\ f}$
   (c) $\overline{\Gamma \vdash T_f <: T_{decl}}$

(3) Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta' \vdash_l () : \mathsf{unit} \dashv \Delta'$. Note that every T_f is a subtype of T_{decl}. The impact on $refTypes(\Sigma', \Delta', o)$ is that types defined for fields will replace types defined in Δ. But because every replacement is a supertype of the type that it replaces, we have Γ, Σ', Δ' **ok** by the *subtype compatibility* lemma (C.3). □

Theorem C.3 (Asset retention). *Suppose:*

(1) Γ, Σ, Δ ***ok***
(2) $o \in dom(\mu)$
(3) $refTypes(\Sigma, \Delta, o) = \overline{D}$
(4) $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$
(5) e *is closed*
(6) $\Sigma, e \to \Sigma', e'$
(7) $refTypes(\Sigma', \Delta', o) = \overline{D'}$
(8) $\exists T' \in \overline{D}$ *such that* $\Gamma \vdash nonDisposable(T')$
(9) $\forall T' \in \overline{D'} : \Gamma \vdash disposable(T')$

Then in the context of a well-typed program, either $\Gamma \vdash nonDisposable(T)$ *or* $e = \mathbb{E}[disown\ \ s]$, *where* $\rho(s) = o$.

Proof. By induction on the typing derivation.

Case: T-lookup. In (6), the only rule that could have applied is E-lookup, which leaves Σ unchanged. Δ' is the same as Δ except that some instances of T_1 have been replaced by T_3. If $\Gamma \vdash nonDisposable(T)$, then it is proved. Otherwise, $\Gamma \vdash disposable(T)$, and by the definition of split, $\Gamma \vdash disposable(T_1)$ and $\Gamma \vdash disposable(T_3)$, so there was no change in disposability in Δ', contradicting the conjunction of (8) and (9).

Case: T-Assign. By assumption, $\Gamma; \Delta, s' : T_{s'}, s'' : T_{s''} \vdash_s s' := s'' : \mathsf{unit} \dashv \Delta'', s' : T^*, s'' : T^{**}$. By inversion, $\Gamma \vdash disposable(T'_s)$, so no owned references to assets were lost by replacing T'_s. As in the case for T-lookup, the definition of split (by inversion, $T_{s''} \Rrightarrow T^*/T^{**}$) ensures that either (8) or (9) is contradicted.

Case: T-Let. $e = \mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2$. There are two subcases, depending on the rule that was used for $\Sigma, e \to \Sigma', e'$:

Subcase: E-let. Σ' has a new mapping for a new indirect reference l, which may cause an additional alias to an object, but all previous aliases are preserved, so it cannot be the case that all non-disposable references are gone.

Subcase: E-letCongr. The induction hypothesis applies to e_1, because $\Sigma, e_1 \to \Sigma', e'_1$. This suffices to prove the case, because there are no changes to Δ.

Case: T-new. By rule E-New, $\Sigma, \mathsf{new}\ C\langle\overline{T'}\rangle@S(\overline{l}) \to [\mu[o \mapsto C\langle\overline{T'}\rangle@S(\overline{\rho(l)})]/\mu]\,\Sigma, o$. By inversion, $\Gamma; \Delta \vdash_s \overline{s'} : \overline{T} \dashv \Delta'$. By the induction hypothesis, any nondisposable references in Δ are preserved in Δ'. The new Σ' also preserves any existing nondisposable references.

Case: T-this-field-def. Rule *E-field* leaves Σ unchanged. By the *split non-disposability* lemma (C.2), if $\Gamma \vdash disposable(T_1)$, then $\Gamma \vdash disposable(T_3)$. No other types are changed in the typing context.

Case: T-this-field-ctxt. Same argument as for *This-field-def*.

Case: T-fieldUpdate. Although Σ' replaces a field, which may reference an object, the reference that was overwritten was disposable (by inversion).

Case: T-inv. The changes in Δ consist of replacing types with the results of $funcArg$. Σ' has additional aliases to objects, but additional aliases cannot cause loss of owning references. We consider the cases for $funcArg$:

FuncArg-owned-unowned. This case preserves ownership in the output type.
FuncArg-shared-unowned. The input type here is not owned.
FuncArg-other. If $owned(T_C@T_{STinput-decl})$, then in Δ, the corresponding variable is an owning type. By substitution, ownership of the object will be maintained in the next context.
This represents a contradiction with the assumption that ownership was lost.
Case: T-privInv. This case is analogous to the case for Public-Invoke, but with additional aliases changed due to fields.

Case: T-$\rightarrow_p$. $e = s \rightarrow_p S'(\overline{x})$. A rule $E- \rightarrow_p$ applied, replacing an object that previously had a type consistent with $C\langle\overline{T_A}\rangle@T_{ST}$ in μ with one that references an object in state S'. The new static context contains an owning reference to the new object, so ownership of s was not lost. For the dynamic context Σ', it suffices to examine the references from fields of the old object ($\mu(\rho(l))$). It remains to consider the fields that were overwritten, but these all had types that were disposable (by inversion of T-$\rightarrow_p$).

Case: T-assertStates. This rule causes no change in either Δ or Σ, which is a contradiction.

Case: T-assertPermission. This rule causes no change in either Δ or Σ, which is a contradiction.

Case: T-assertInVar. This rule causes no change in either Δ or Σ, which is a contradiction.

Case: T-assertInVarAlready. This rule causes no change in either Δ or Σ, which is a contradiction.

Case: T-IsIn-StaticOwnership. $e = \mathsf{if}\ x\ \mathsf{in}_{owned}\ S\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$.

(1) If E-IsIn-Owned-Then applies, then $\Sigma, e \rightarrow \Sigma, e_1$ (and by the preservation lemma, e_1 is well-typed). By the same argument as for T-lookup, no ownership was lost in Δ' and Δ''; any consumed ownership is now in T_1. From the *merging preserves nondisposability* lemma (C.21), we find a contradiction with the assumption that a type has changed from nondisposable to disposable in this step.
(2) Otherwise, E-IsIn-Else applies, and the same argument applies to e_2.

Case: T-isInDynamic. $e = \mathsf{if}\ x\ \mathsf{in}_{shared}\ S\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. The same argument as in the T-IsIn-StaticOwnership case applies, except that the situation is even simpler, because Δ and Δ' agree that $x : T_C@Shared$.

Case: T-IsIn-PermVar. The argument is the same as for T-isInDynamic.

Case: T-IsIn-Perm-Then. E-IsIn-Perm-Then applies, and, $\Sigma' = \Sigma$. By the same argument as in the T-Lookup case, no ownership was lost in Δ', which contradicts the assumption.

Case: T-IsIn-Perm-Else. The argument is the same as for T-IsIn-Perm-Then, but with E-IsIn-Perm-Else.

Case: T-IsIn-Unowned. The argument is the same as for T-IsIn-Perm-Then, but with E-IsIn-Unowned.

Case: T-disown. Then $e = \mathsf{disown}\ \ s$.

Case: T-pack. Note that pack leaves Σ unchanged; the only change is removing $\overline{s.f : T_f}$ from Δ. But by inversion, $\overline{T_f \approx T_{decl}}$. As a result, no ownership can change from Δ to Δ', contradicting the assumptions.

Case: T-state-mutation-detection. $e = \boxed{e'}_o$. The step must have been either via E-Box-ϕ or via E-Box-ϕ-congr.

Case: E-Box-ϕ. The change in E-Box-ϕ and state-mutation-detection has no impact on ownership, so this contradicts the assumptions.

Case: E-Box-ϕ-congr. We have the required property by the induction hypothesis, since the present rules make no changes themselves to Δ' and Σ', which were provided inductively.

Case: T-reentrancy-detection. $e = \boxed{e'}^{o}$. The step must have been either via E-Box-ψ or via E-Box-ψ-congr.

Case: E-Box-ψ. The change in E-Box-ψ and state-mutation-detection has no impact on ownership, so this contradicts the assumptions.

Case: E-Box-ψ-congr. We have the required property by the induction hypothesis, since the present rules make no changes themselves to Δ' and Σ', which were provided inductively. □

C.1 Supporting Lemmas

Lemma C.1 (Memory consistency). *If* Γ, Σ, Δ ***ok****, then:*

(1) If $l : C\langle\overline{T'}\rangle@S \in \Delta$*, then* $\exists o.\rho(l) = o$ *and* $\mu(o) = C\langle\overline{T'}\rangle@S(\overline{s})$.
(2) If $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$*, and* l *is a free variable of* e*, then* $l \in dom(\rho)$.

Proof.

(1) Assume $l : C\langle\overline{T'}\rangle@S \in \Delta$. Then $\rho(l) = o$ follows by inversion of global consistency. $\mu(o) = C^*\langle\overline{T^*}\rangle@S'(\overline{o'})$ follows by inversion of reference consistency (which itself follows by inversion of global consistency). By inversion of reference consistency, $\cdot \vdash C^*\langle\overline{T^*}\rangle@S' <: \overline{D}$. By definition of refTypes, $C\langle\overline{T'}\rangle@S \in \overline{D}$, so $\cdot \vdash C^*\langle\overline{T^*}\rangle@S' <: C\langle\overline{T'}\rangle@S$. This implies that $C = C^*$, $S = S'$, and $\cdot \vdash \overline{T^*} <: \overline{T'}$ (by definition of subtyping).
(2) By induction on the typing derivation, we prove that if l is a free variable of e, then $l \in dom(\Delta)$. Then the conclusion follows immediately from the definition of global consistency. We consider some example cases:
Case: T-lookup. s' is a free variable, but $s' : T_1 \in \Delta$.
Case: T-let. Any free variables in e must be in e_1 or e_2. The result is obtained by induction on e_1 and e_2.
Case: $s \rightarrow_p S'(\overline{x})$. s is a free variable, but $s : C\langle\overline{T_A}\rangle@T_{ST} \in \Delta$.
Case: T-assertStates. x is a free variable, but $x \in dom(\Delta)$.
The remaining cases are similar to the above. □

Lemma C.2 (Split Non-disposability). *If* $T_1 \Rrightarrow T_2/T_3$*, and* T_1 *is not disposable, then* T_2 *is not disposable.*

Proof. By inspection of the definition of $T_1 \Rrightarrow T_2/T_3$ and *owned*. Note that in the *Split-owned-shared* and *Split-states-shared* cases, although $owned(T_1)$, C is not an asset, which makes T_1 disposable. □

Lemma C.3 (Subtype Compatibility). *If* $T \leftrightarrow T'$*, and* $\Gamma \vdash T' <: T''$*, then* $T \leftrightarrow T''$.

Proof. By straightforward case analysis of the subtyping relation. □

Lemma C.4 (Subtyping reflexivity). *For all types* T, $\Gamma \vdash T <: T$.

Proof.

Case: unit. Rule Unit applies.
Case: $T_C@T_{ST}$. By rule Refl in the definition of the subpermission relation, rule *Matching-definitions* applies. □

Lemma C.5 (Exclusivity of isAsset/nonAsset). *For all types* T*:*

(1) If $\Gamma \vdash isAsset(T)$ *is provable, then* $\Gamma \vdash nonAsset(T)$ *is not provable.*
(2) If $\Gamma \vdash nonAsset(T)$ *is provable, then* $\Gamma \vdash isAsset(T)$ *is not provable.*

Proof. By straightforward case analysis of the isAsset and nonAsset rules. □

Lemma C.6 (Exclusivity of isVar/nonVar). *For all types T:*

(1) If isVar (T) is provable, then nonVar (T) is not provable.
(2) If nonVar (T) is provable, then isVar (T) is not provable.

For all declaration types T_C:

(1) If isVar (T_C) is provable, then nonVar (T_C) is not provable.
(2) If nonVar (T_C) is provable, then isVar (T_C) is not provable.

For all permissions/states T_{ST}:

(1) If isVar (T_{ST}) is provable, then nonVar (T_{ST}) is not provable.
(2) If nonVar (T_{ST}) is provable, then isVar (T_{ST}) is not provable.

Proof. By straightforward case analysis of the isVar and nonVar rules. □

Lemma C.7 (Exclusivity of maybeOwned/notOwned). *For all types T:*

(1) If maybeOwned (T) is provable, then notOwned (T) is not provable.
(2) If notOwned (T) is provable, then maybeOwned (T) is not provable.

Proof. By straightforward case analysis of the ownedState and notOwned rules. □

Definition C.1 (Non-disposability).

$$\frac{\mathit{maybeOwned}(T_C@T_{ST}) \qquad \Gamma \vdash \mathit{isAsset}(T_C@T_{ST})}{\Gamma \vdash \mathit{nonDisposable}(T_C@T_{ST})}\ \text{ND-owned.}$$

Lemma C.8 (Exclusivity of disposability and non-disposability). *For all types T:*

(1) If $\Gamma \vdash$ disposable (T) is provable, then $\Gamma \vdash$ nonDisposable (T) is not provable.
(2) If $\Gamma \vdash$ nonDisposable (T) is provable, then $\Gamma \vdash$ disposable (T) is not provable.

Proof.

(1) Consider the cases for $\Gamma \vdash$ *disposable* (T).
Case: D-Owned. Let $T = T_C@T_{ST}$. By inversion, *maybeOwned* $(T_C@T_{ST})$ and $\Gamma \vdash$ *nonAsset* $(T_C@T_{ST})$. Then we cannot prove *nonDisposable*, which requires $\Gamma \vdash$ *isAsset* $(T_C@T_{ST})$.
Case: D-not-owned. There is no rule by which to prove $\Gamma \vdash$ *nonDisposable* (T).
Case: D-Unit. There is no rule by which to prove $\Gamma \vdash$ *nonDisposable* (T).
(2) To prove $\Gamma \vdash$ *nonDisposable* (T), we must use ND-Owned; so $T = T_C@T_{ST}$, and we must show that *maybeOwned* $(T_C@T_{ST})$ and $\Gamma \vdash$ *isAsset* $(T_C@T_{ST})$. But this directly contradicts the premises of D-not-owned and D-owned, and D-unit does not apply. So there is no rule by which to prove $\Gamma \vdash$ *disposable* (T). □

Lemma C.9 (Interface substitution). *If*

(1) $\Gamma; \Delta, s' : I\langle\overline{T}\rangle@T_{ST} \vdash_s e : T \dashv \Delta'$
(2) $\Gamma \vdash C\langle\overline{T'}\rangle <: I\langle\overline{T}\rangle$
*(3) C **ok***

then $\Gamma; \Delta, s' : C\langle\overline{T'}\rangle@T_{ST} \vdash_s e : T \dashv \Delta''$, where

(1) *if* $s' : I\langle\overline{T}\rangle@T'_{ST} \in \Delta'$, *then* $\Delta'' = \Delta', s' : C\langle\overline{T'}\rangle@T'_{ST}$
(2) *otherwise* $\Delta'' = \Delta'$.

Proof. By induction on the typing derivation. The relevant cases are Inv and P-Inv; all other cases will be identical, because x has the same permission or state. Because interfaces do not have fields, there must not be any field assignments or access involving x, so we do not need to consider those.

Case: Inv In this case, $e = s_1.m\langle\overline{T_M}\rangle(\overline{s_2})$.
If $s' = s_1$, with the assumption that m **ok in** C, then we have:
(1) $specializeTrans_\Gamma\left(m\langle\overline{T_M}\rangle, I\langle\overline{T}\rangle\right) = T\ m\langle\overline{T'_M}\rangle(\overline{T_{C_x}@T_x \gg T_{xST}\ x})\ T_{this} \gg\ T'_{this}$
(2) C **ok**
(3) $\Gamma \vdash T_{ST} <:_* T_{this}$
(4) $\overline{\Gamma \vdash T_{s2} <: T_{C_x}@T_x}$
(5) $T'_{s1} = funcArg(T_C@T_{ST\,s1}, T_C@T_{this}, T_C@T'_{this})$
(6) $\overline{T'_{s2} = funcArg(T_{s2}, T_x, T_{C_x}@T_{xST})}$
So then
(1) $transactionName\,(m) \in transactionNames\,(C)$
(2) $def\,(m, C) = M = T'\ m\langle\overline{T'_M}\rangle(\overline{T'_{C_x}.T'_x \gg T'_{xST}\ x})\ T^*_{this} \gg T^{**}_{this}$
(3) $implementOk_\Gamma(I\langle\overline{T}\rangle, M)$.
By definition of implementOk, this implies that the invocation is still well-typed.
If $s' \in \overline{s_2}$, then $\Gamma \vdash I\langle\overline{T}\rangle@T^*_{ST} <: T$ for some argument of type T. But then because subtyping is transitive, $C'\langle\overline{T'}\rangle@T^*_{ST}$ is also a subtype of T, so the invocation is still safe.
Case: P-Inv Identical to the Inv case, except that we cannot invoke a private transaction on s', as interfaces do not have private transactions, so s' must be one of the arguments. □

Lemma C.10 (Permission Variable Substitution). *Suppose*

(1) $\Gamma \vdash T_{ST} <:_* p$
(2) $\Gamma; \Delta, x : T_C@p \vdash_s e : T \dashv \Delta'$

Then $\Gamma; \Delta, x : T_C@T_{ST} \vdash_s e : T \dashv \Delta'$.

Proof. Follows from C.4 □

Lemma C.11 (Exclusivity of subpermission). *For any permissions* P *and* P'*:*

(1) *If* $\cdot \vdash P <:_* P'$ *is provable, then* $\cdot \vdash P \not<:_* P'$ *is not provable.*
(2) *If* $\cdot \vdash P \not<:_* P'$ *is provable, then* $\cdot \vdash P <:_* P'$ *is not provable.*

Proof. By case analysis of the subpermission rules, we can see that every pair of permissions is related. The only way that $\cdot \vdash P <:_* P'$ **and** $\cdot \vdash P' <:_* P$ can be true is if $P = P'$, but then we cannot prove $\cdot \vdash P \not<:_* P'$. □

Lemma C.12 (Split compatibility). *If* $\Gamma; \Delta \vdash_s \overline{s'} : \overline{T} \dashv \Delta'$ *and* Γ, Σ, Δ**ok**, *then* Γ, Σ, Δ' **ok**.

Proof. For one expression, it suffices to show that replacing T with T_3 in Δ leaves the remaining context consistent with Σ. The proof of this is by cases of splitting; this is theorem `splittingRespectsHeap` in heapLemmasforSplitting.agda in the supplement. For multiple expressions, simply iterate the argument. □

Lemma C.13 (Substitution). *If* $\Gamma; \Delta, x : T_x \vdash_s e : T' \dashv \Delta', x : T'_x$, *then* $\Gamma; \Delta, l : T_x \vdash_s [l/x]e : T' \dashv \Delta', l : T'_x$.

Proof. Substitute l for x throughout the previous proof. □

Lemma C.14 (Subtype replacement). *If*

- $\Gamma; \Delta, x : T_x \vdash_s e : T' \dashv \Delta', x : T'_x$
- $\Gamma \vdash T''_x <: T_x$
- $T''_x \approx T_x$

then $\Gamma; \Delta, x : T''_x \vdash_s e : T' \dashv \Delta', x : T'''_x$ *where* $\Gamma \vdash T'''_x <: T'_x$.

Proof. By induction on the typing derivation and the subtyping derivation. Relevant cases include:

Case: T-lookup.
(1) By assumption:
  (a) $\Gamma \vdash T''_x <: T_x$
(2) By inversion of T-lookup:
  (a) $T_x \Rrightarrow T'/T'_x$
(3) Note that it suffices to show that $T''_x \Rrightarrow T'/T'''_x$. Consider the cases for 2a:
  Case: Split-unowned $T'_x = T_C$@Unowned. Split-Unowned applies to T''_x, resulting in $T'''_x = T_C$@Unowned.
  Case: Split-shared By assumption, $T_x = T_C$@Shared. If $T''_x = T_C$@Shared, then the result follows by Split-Shared. Otherwise, *maybeOwned* (T''_x), but this contradicts the assumption that $T''_x \approx T_x$.
  Case: Split-owned-shared By inversion of maybeOwned, we have the following cases:
    Subcase: $T_x = T_C$@p. All subtypes of T_C@p are themselves maybeOwned and nonAsset, so Split-owned-shared applies.
    Subcase: $T_x = T_C$@Owned. All subtypes of T_C@Owned are themselves maybeOwned and nonAsset, so Split-owned-shared applies.
    Subcase: $T_x = T_C$@$\overline{S}$. All subtypes of T_C@$\overline{S}$ are themselves maybeOwned and nonAsset, so Split-owned-shared applies.
  Case: Split-unit. $T_x = T'_x =$ unit. Split-unit applies for T''_x, since the only subtype of unit is unit. Then $T'''_x =$ unit, which is a subtype of T'_x.

Case: T-IsIn-StaticOwnership. $\Gamma \vdash T''_x <: T_x$ results in a smaller set of initial possible states for x, resulting in a potentially smaller set of possible states for x in the resulting context. This explains why it is not necessarily the case that $T'''_x = T''_x$. □

Corollary C.4 (Subtype substitution). *If*

- $\Gamma; \Delta, x : T_x \vdash_s e : T' \dashv \Delta', x : T'_x$ *and*
- $\Gamma \vdash T''_x <: T_x$

then $\Gamma; \Delta, l : T''_x \vdash_s [l/x]e : T' \dashv \Delta', l : T'''_x$ *where* $\Gamma \vdash T'''_x <: T'_x$.

Proof. Follows by applying both C.14 and C.13. □

Corollary C.5 (l-stronger substitution). *If* $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$ *and* $\Delta'' <^l_{\Gamma,\Sigma'} \Delta$, *then* $\Gamma; \Delta'' \vdash_s e : T \dashv \Delta'''$ *with* $\Delta''' <^l_{\Gamma,\Sigma'} \Delta'$.

Proof. By induction on Δ', applying C.4 and the definition of $\Delta'' <^l_{\Gamma,\Sigma'} \Delta$. □

Lemma C.15 (l-stronger consistency). *If* $\Delta' <^l_{\Gamma,\Sigma'} \Delta$ *and* Γ, Σ, Δ**ok**, *then* Γ, Σ, Δ' **ok**.

Proof. By induction on Δ and application of subtype compatibility (C.3). □

Lemma C.16 (Strengthening). *If* $\Gamma; \Delta, s' : T_0 \vdash_s e : T \dashv \Delta', s' : T_1$, *and* s' *does not occur free in* e, *then* $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$.

Proof. By induction on the typing derivation. Since s' does not occur free in e, s' must not be needed in either proof. □

Lemma C.17 (Weakening). *If* $\Gamma; \Delta \vdash_s e : T \dashv \Delta'$, *and* s' *does not occur free in* e, *then* $\Gamma; \Delta, s' : T_0 \vdash_s e : T \dashv \Delta', s' : T_1$.

Proof. By induction on the typing derivation. Since s' does not occur free in e, s' must not be needed in either proof. □

Lemma C.18 (Merge consistency). *If* Γ, Σ, Δ ***ok*** *and* Γ, Σ, Δ' ***ok***, *then* $\Gamma, \Sigma, merge(\Delta, \Delta')$ ***ok***.

Proof. By induction on $merge(\Delta, \Delta')$.

Case: Sym. By the induction hypothesis, $\Gamma, \Sigma, merge(\Delta', \Delta)$**ok**, and $merge(\Delta', \Delta) = merge(\Delta, \Delta')$.

Case: $\oplus$. By inversion, $\Delta = \Delta'', x : T$ and $\Delta' = \Delta''', x : T'$. Because Δ'' is a subset of $\Delta'', x : (T \oplus T')$, by the induction hypothesis, $\Gamma, \Sigma, merge(\Delta'', \Delta''')$ **ok** (the induction hypothesis applies, because $dom(\Delta'') \subset dom(\Delta'', x : (T \oplus T'))$ and $\Delta''(x') = \Delta(x')$ for all $x \neq x'$). Therefore, by the definition of consistency, it suffices to show that $T \oplus T'$ is compatible with all $T'' \in refTypes(\Sigma, \Delta'')$. We assume that either $T'' \leftrightarrow T$ or $T'' \leftrightarrow T'$.

Subcase: $T \oplus T = T$. Anything compatible with T is trivially compatible with T.

Subcase: T_C@*Owned* $\oplus$ T_C@$\overline{S}$ = T_C@*Owned*. If T'' is compatible with $T_C@Owned$, then it is proved. Otherwise, T'' is compatible with $T_C@\overline{S}$ (by inspection of the definition of $\leftrightarrow$). In particular, T'' must be $T_C@Unowned$, in which case rule UnownedOwnedCompat applies.

Subcase: T_C@*Shared* $\oplus$ T_C@*Unowned* = T_C@*Unowned*. If T'' is compatible with $T_C@Unowned$, then it is proved. Otherwise, T'' is compatible with $T_C@Shared$, and by definition of $\leftrightarrow$, either $T'' = T_C@Shared$ or $T'' = T_C@Unowned$. The later case was already addressed, and in the former case, SharedCompat gives $T'' \leftrightarrow T_C@Shared$.

Subcase: $T_C@\overline{S} \oplus T_C@\overline{S'} = T_C@(S \cup S')$. The only compatibility rule that could have applied was UnownedStatesCompat, and it still applies to $T_C@(S \cup S')$.

Subcase: $C\langle\overline{T}\rangle@T_{ST} \oplus I\langle\overline{T'}\rangle@T'_{ST} = I\langle\overline{T \oplus T'}\rangle@T_{ST} \oplus I\langle\overline{T \oplus T'}\rangle@T'_{ST} = I\langle\overline{T^*}\rangle@T^*_{ST}$. If T'' is compatible with $C\langle\overline{T}\rangle@T_{ST}$, then it will also be compatible with $I\langle\overline{T^*}\rangle@T^*_{ST}$ by SubtypeCompat, ParamCompat, and application of one of the other subcases for T^*_{ST}. If T'' is compatible with $I\langle\overline{T'}\rangle@T'_{ST}$, then it will also be compatible with $I\langle\overline{T^*}\rangle@T^*_{ST}$ by ParamCompat and application of one of the other subcases for T^*_{ST}.

Subcase: $D\langle\overline{T}\rangle@T_{ST} \oplus D\langle\overline{T'}\rangle@T'_{ST} = D\langle\overline{T \oplus T'}\rangle@T_{ST} \oplus D\langle\overline{T \oplus T'}\rangle@T'_{ST} = D\langle\overline{T^*}\rangle@T^*_{ST}$. If T'' is compatible with $D\langle\overline{T}\rangle@T_{ST}$, then it will also be compatible with $D\langle\overline{T^*}\rangle@T^*_{ST}$ by ParamCompat, and application of one of the other subcases for T^*_{ST}. If T'' is compatible with $D\langle\overline{T'}\rangle@T_{ST}$, then it will also be compatible with $D\langle\overline{T^*}\rangle@T^*_{ST}$ by ParamCompat, and application of one of the other subcases for T^*_{ST}.

Case: Dispose-disposable. Eliminating a variable from a context that is already consistent with Σ leaves a context that is still consistent with Σ. Note that this rule does not allow removing bindings of the form $x.f : T$, because removing those bindings could result in inconsistencies, since then the types of those fields would (incorrectly) be assumed to be according to their declarations. □

Lemma C.19 (Merge Subtyping). *If* $l : T \in merge(\Delta^*, \Delta^{**})$, *then* $l : T_1 \in \Delta^*$ *and* $l : T_2 \in \Delta^{**}$ *with* $\Gamma \vdash T_1 <: T$, $\Gamma \vdash T_2 <: T$, $T_1 \approx T$, *and* $T_2 \approx T$.

Proof. By induction on the merge judgment.

Case: Sym. The conclusion follows immediately from the induction hypothesis.
Case: $\oplus$. In each subcase, the conclusion follows from the induction hypothesis and the $\oplus$ subtyping lemma (C.20).
Case: Dispose-disposable. $d \notin merge(\Delta^*, \Delta^{**})$, so the conclusion follows immediately from the induction hypothesis. □

Lemma C.20 ($\oplus$ subtyping). *If* $T_1 \oplus T_2 = T$, *then* $\Gamma \vdash T_1 <: T$ *and* $\Gamma \vdash T_2 <: T$.

Proof.

Case: $T \oplus T$. It is proved by reflexivity of <:.
Case: $T_C@Owned \oplus T_C@\overline{S}$. $\Gamma \vdash T_C@Owned <: T_C@Owned$ and $\Gamma \vdash T_C@\overline{S} <: T_C@Owned$.
Case: $T_C@Shared \oplus T_C@Unowned$. $\Gamma \vdash T_C@Shared <: T_C@Unowned$ and $\Gamma \vdash T_C@Unowned <: T_C@Unowned$.
Case: $T_C@\overline{S} \oplus T_C@\overline{S'}$. $\Gamma \vdash T_C@\overline{S} <: T_C@(S \cup S')$ and $\Gamma \vdash T_C@\overline{S'} <: T_C@(S \cup S')$.
Case: $C\langle\overline{T}\rangle@T_{ST} \oplus I\langle\overline{T}\rangle@T'_{ST}$. $\Gamma \vdash C\langle\overline{T}\rangle@T_{ST} <: I\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$ and $\Gamma \vdash I\langle\overline{T}\rangle@T'_{ST} <: I\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$ by rule *implements-interface* and the induction hypothesis.
Case: $D\langle\overline{T}\rangle@T_{ST} \oplus D\langle\overline{T}\rangle@T'_{ST}$. $\Gamma \vdash D\langle\overline{T}\rangle@T_{ST} <: D\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$ and $\Gamma \vdash D\langle\overline{T}\rangle@T'_{ST} <: D\langle\overline{T}\rangle.(T_{ST} \oplus T'_{ST})$ by rule Matching-Declarations and the induction hypothesis.

Note that in each of the above cases, $T_1 \approx T_2$. □

Theorem C.6 (Unicity of ownership). *If* Γ, Σ, Δ ***ok***, *and* $o \mapsto C\langle\overline{T}\rangle@S(\ldots) \in \mu$, *and* $refTypes(\Sigma, \Delta, o) = \overline{D}$, *then at most one* $T \in \overline{D}$ *is either* $C\langle\overline{T}\rangle.\overline{S}$ *or* $C\langle\overline{T}\rangle@Owned$.

Proof. By inversion of reference consistency, $\forall T_1, T_2 \in \overline{D}, T_1 \leftrightarrow T_2$ or ($o \in \Sigma_\phi$ and $T_i = C\langle\overline{T}\rangle@S$ and $T_j = C\langle\overline{T}\rangle@Shared (i \neq j)$). Note that $C\langle\overline{T}\rangle@Owned$ is not compatible with either $C\langle\overline{T}\rangle@Owned$ or $C\langle\overline{T}\rangle.\overline{S}$, and $C\langle\overline{T}\rangle.\overline{S}$ is not compatible with $C\langle\overline{T}\rangle.\overline{S}$. If there were more than one alias of type $C\langle\overline{T}\rangle@Owned$ or $C\langle\overline{T}\rangle.\overline{S}$, then they would be incompatible, which would be a contradiction. Even if $o \in \Sigma_\phi$, the aliases are restricted to shared and state-specifying aliases, and never more than one state-specifying alias exists. □

Lemma C.21 (Merging preserves nondisposability). *Suppose* Δ_1, Δ_2 *are static contexts. If* ($s : T \in \Delta_1$ *or* $s : T \in \Delta_2$), $\Gamma \vdash nonDisposable(T)$, *and* $merge(\Delta_1, \Delta_2) = \Delta$, *then* $s : T' \in \Delta$ *such that* $\Gamma \vdash nonDisposable(T')$.

Proof. By case analysis on $merge(\Delta_1, \Delta_2)$.

Case: Sym. The induction hypothesis applies to $merge(\Delta, \Delta')$ since the lemma was stated symmetrically.
Case: $\oplus$. Note that in all cases of the definition of $T_1 \oplus T_2 = T_3$, if either $owned(T_1)$ or $owned(T_2)$, then $owned(T_3)$as well.
Case: Dispose-disposable. Without loss of generality, suppose $s : T \in \Delta_1$. By inversion, $x \notin \Delta_2$. By assumption, $\Gamma \vdash nonDisposable(T)$. But by inversion, $\Gamma \vdash disposable(T)$. This is a contradiction (C.8). □

ACKNOWLEDGMENT

We appreciate the help of Eliezer Kanal at the Software Engineering Institute, who helped start this project; Jim Laredo, Rick Hull, Petr Novotny, and Yunhui Zheng at IBM, who provided useful technical and real-world insight; and David Gould and Georgi Panterov at the World Bank, with whom we worked on the insurance case study. We also appreciate the help of the anonymous reviewers, who have helped us refine this research.

REFERENCES

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented Programming. In *Companion of Object Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*. 1015–1022. DOI: https://doi.org/10.1145/1639950.1640073

Leonardo Alt and Christian Reitwiessner. 2018. SMT-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*.

Tara Astigarraga, Xiaoyan Chen, Yaoliang Chen, Jingxiao Gu, Richard Hull, Limei Jiao, Yuliang Li, and Petr Novotny. 2018. Empowering business-level blockchain users with a rules framework for smart contracts. In *International Conference on Service-Oriented Computing (ICSOC'18)*. DOI: https://doi.org/10.1007/978-3-030-03596-9_8

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts SoK. In *Principles of Security and Trust (POST'17)*. DOI: https://doi.org/10.1007/978-3-662-54455-6_8

Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. 2017. A user study to inform the design of the obsidian blockchain DSL. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'17)*.

Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. 2016. Formal verification of smart contracts. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS'16)*. DOI: https://doi.org/10.1145/2993600.2993611

Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: Checking protocol compliance under aliasing. In *Companion of International Conference on Software Engineering (ICSE Companion'08)*. 971–972. DOI: https://doi.org/10.1145/1370175.1370213

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2009. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming (ECOOP'09)*. DOI: https://doi.org/10.1007/978-3-642-03013-0_10

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2011. Checking concurrent typestate with access permissions in Plural: A retrospective. In *Engineering of Software*, P. Tarr and A. Wolf (Eds.). Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-642-19823-6_4

John Boyland. 2003. Checking interference with fractional permissions. In *International Conference on Static Analysis (SAS'03)*. DOI: https://doi.org/10.1007/3-540-44898-5_4

John Boyland, James Noble, and William Retert. 2001. Capabilities for sharing: A generalisation of uniqueness and read-only. In *European Conference on Object-Oriented Programming (ECOOP'01)*. DOI: https://doi.org/10.1007/3-540-45337-7_2

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory (CONCUR'10)*. DOI: https://doi.org/10.1007/978-3-642-15375-4_16

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. DOI: https://doi.org/10.1145/286936.286947

David G. Clarke, Tobias Wrigstad, and James Noble. 2013. *Aliasing in Object-oriented Programming: Types, Analysis and Verification*. Lecture Notes in Computer Science, Vol. 7850. Springer. DOI: https://doi.org/10.1007/978-3-642-36946-9

Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2014. Considering productivity effects of explicit type declarations. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'14)*. 3. DOI: https://doi.org/10.1145/2688204.2688218

Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020b. Obsidian smart contract programming language. Carnegie Mellon University. DOI: https://doi.org/10.1184/R1/12814202.v1

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary programming language design. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'18)*. 133–146. DOI: https://doi.org/10.1145/3276954.3276965

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020a. Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in obsidian. In *Object-oriented Programming Systems, Languages, and Applications (OOPSLA'20)*. Submitted for publication.

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019a. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. arxiv:1912.04719. Retrieved from http://arxiv.org/abs/1912.04719.

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019b. Smarter smart contract development tools. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. DOI: https://doi.org/10.1109/WETSEB.2019.00013

Phil Daian. 2016. Analysis of the DAO exploit. Retrieved August 21, 2018 from http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

Robert DeLine and Manuel Fähndrich. 2004. Typestates for objects. In *European Conference on Object-Oriented Programming (ECOOP'04)*. DOI: https://doi.org/10.1007/978-3-540-24851-4_21

Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. DOI: https://doi.org/10.1007/978-3-662-53357-4_6

Vincent Dieterich, Marko Ivanovic, Thomas Meier, Sebastian Zäpfel, Manuel Utz, and Philipp Sandner. 2017. Retrieved February 18, 2020 from https://medium.com/@philippsandner/application-of-blockchain-technology-in-the-manufacturing-industry-d03a8ed3ba5e.

Digital Asset, Inc. 2019. An Introduction to DAML. Retrieved February 18, 2020 from https://docs.daml.com/daml/intro/0_Intro.html.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2002. More dynamic object reclassification: Fickle II. *ACM Trans. Program. Lang. Syst.* 24, 2 (Mar. 2002), 153–191. DOI: https://doi.org/10.1145/514952.514955

Chris Elsden, Arthi Manohar, Jo Briggs, Mike Harding, Chris Speed, and John Vines. 2018. Making sense of blockchain applications: A typology for HCI. In *CHI Conference on Human Factors in Computing Systems (CHI'18)*. 1–14. DOI: https://doi.org/10.1145/3173574.3174032

Encyclopædia Britannica. 2020. Obsidian. Retrieved May 24, 2020 from https://www.britannica.com/science/obsidian.

Ethereum Foundation. 2020c. Common Patterns. Retrieved February 18, 2020 from http://solidity.readthedocs.io/en/develop/common-patterns.html.

Ethereum Foundation. 2020b. Ethereum Project. Retrieved February 18, 2020 from http://www.ethereum.org.

Ethereum Foundation. 2020a. Solidity. Retrieved February 18, 2020 from https://solidity.readthedocs.io/en/develop/.

Manuel Fahndrich and Robert DeLine. 2002. Adoption and focus: Practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI'02)*. 12. DOI: https://doi.org/10.1145/512529.512532

J. Feist, G. Grieco, and A. Groce. 2019. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (October 2014), 44 pages. DOI: https://doi.org/10.1145/2629609

Google Inc. 2019. Protocol Buffers. Retrieved February 18, 2020 from https://developers.google.com/protocol-buffers/.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Object-oriented Programming, Systems, Languages, and Applications* (2012). DOI: https://doi.org/10.1145/2398857.2384619

Luke Graham. 2017. $32 million worth of digital currency ether stolen by hackers. Retrieved November 2, 2017 from https://www.cnbc.com/2017/07/20/32-million-worth-of-digital-currency-ether-stolen-by-hackers.html.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: Analyzing the out-of-gas world of smart contracts. *Commun. ACM* 63, 6 (2020).

Harvard Business Review. 2017. The Potential for Blockchain to Transform Electronic Health Records. Retrieved February 18, 2020 from https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records.

Dominik Harz and William Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. arxiv:1809.09805. Retrieved from http://arxiv.org/abs/1809.09805.

Richard Hull, Vishal S. Batra, Yi-Min Chen, Alin Deutsch, Fenno F. Terry Heath III, and Victor Vianu. 2016. Towards a shared ledger business collaboration language based on data-aware processes. In *International Conference on Service-Oriented Computing (ICSOC'16)*.

IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from https://www.ibm.com/blockchain/supply-chain/.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450.

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium (NDSS'18)*.

Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Şerbănuţă, Yi Zhang, Daniele Filaretti, Virgil Şerbănuţă, Ralph Johnson, and Grigore Roşu. 2019. IELE: A rigorously designed language and tool ecosystem for the blockchain. In *International Symposium on Formal Methods (FM'19)*.

H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. DOI: https://doi.org/10.1145/319566.319567

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Computer and Communications Security (CCS'16)*. DOI: https://doi.org/10.1145/2976749.2978309

Robert C. Martin, Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education. 95022672

Leonid Mikhajlov and Emil Sekerinski. 1998. A study of the fragile base class problem. In *European Conference on Object-Oriented Programming (ECOOP 1998)*. 355–382.

Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (July 2016), 44–52. DOI: https://doi.org/10.1109/MC.2016.200

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *Principles of Programming Languages (POPL'12)*. DOI: https://doi.org/10.1145/2103621.2103722

Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *SIGCHI Conference on Human Factors in Computing Systems (CHI 1990)*.

John F. Pane, Brad A. Myers, and Leah B. Miller. 2002. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments (HCC'02)*. 198–206. DOI: https://doi.org/10.1109/HCC.2002.1046372

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

Mozilla Research. 2015. The Rust Programming Language. Retrieved February 18, 2020 from https://www.rust-lang.org.

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *J. Logic Algebr. Program.* 79, 6 (2010), 397–434.

Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style. In *Conference on LISP and Functional Programming (LFP'92)*. 11. DOI: https://doi.org/10.1145/141471.141563

Franklin Schrans and Susan Eisenbach. 2019. Introduce the Asset trait. Retrieved February 18, 2020 from https://github.com/flintlang/flint/blob/master/proposals/0001-asset-trait.md.

Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for safer smart contracts. arxiv:1904.06534. Retrieved from https://arxiv.org/abs/1904.06534.

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'19)*. DOI: https://doi.org/10.1145/3360611

Emin Gün Sirer. 2016. Thoughts on The DAO Hack. Retrieved February 18, 2020 from http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/.

Andreas Stefik and Stefan Hanenberg. 2014. The programming language wars: Questions and responsibilities for the programming language community. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2014)*. 283–299. DOI: https://doi.org/10.1145/2661136.2661156

Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ.* 13, 4 (2013), 19.

Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* SE-12, 1 (1986), 157–171. DOI: https://doi.org/10.1109/TSE.1986.6312929

Jeffrey Stylos and Steven Clarke. 2007. Usability implications of requiring parameters in objects' constructors. In *International Conference on Software Engineering (ICSE'07)*. DOI: https://doi.org/10.1109/ICSE.2007.92

Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2014. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *European Conference on Object-Oriented Programming (ECOOP'14)*. DOI: https://doi.org/10.1007/978-3-662-44202-9_7

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *Object Oriented Programming Systems, Languages, and Applications (OOPSLA'11)*. DOI: https://doi.org/10.1145/2076021.2048122

Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997). DOI: https://doi.org/10.5210/fm.v2i9.548

The Linux Foundation. 2020. Hyperledger Fabric. Retrieved February 18, 2020 from https://www.hyperledger.org/projects/fabric.

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Principles of Programming Languages (POPL'11)*. DOI: https://doi.org/10.1145/1926385.1926436

Fabian Vogelsteller and Vitalik Buterin. 2015. EIP 20: ERC-20 Token Standard. Retrieved February 18, 2020 from https://eips.ethereum.org/EIPS/eip-20.

Philip Wadler. 1990. Linear types can change the world. In *Programming Concepts and Methods*, Vol. 2. 347–359.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2017. Design and Implementation of Concurrent C0. arxiv:cs.PL/1701.04929. Retrieved from https://arxiv.org/abs/1701.04929.

Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. 2017. A taxonomy of blockchain-based systems for architecture design. In *International Conference on Software Architecture (ICSA'17)*.
Jakub Zakrzewski. 2018. Towards verification of ethereum smart contracts: A formalization of core of solidity. In *Verified Software. Theories, Tools, and Experiments*.

Received August 2019; revised July 2020; accepted August 2020