User-Centered Design of Principled Programming Languages

Michael J. Coblenz

CMU-CS-20-127

August 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Co-Chair)
Brad A. Myers (Co-Chair)
Frank Pfenning
Joshua Sunshine
Gail C. Murphy (University of British Columbia)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Security Agency Department of Defense award H9823018D0008; by the National Science Foundation awards CNS1423054 and CCF1901033; by the United States Air Force Office of Scientific Research award FA8702-15-D-0002; by two IBM PhD Fellowship awards; and by Ripple. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: user-centered programming language design, usability of programming languages, smart contract languages, immutability, blockchain, empirical studies of programmers

I dedicate this dissertation to my wife, Lauren, who has supported me and encouraged me to pursue this work. I also dedicate this dissertation to my children, Rebecca and Hannah. I hope that the work I do will make the world a better place in which they can live, and I am thankful for their support as I conducted this research.

Abstract

Programming languages exist to enable people to create and maintain software as effectively as possible. They are subject to two very different sets of requirements: first, the need to provide strong safety guarantees to protect against bugs; and second, the need for users to effectively and efficiently write software that meets their functional and quality requirements. This thesis argues that fusing formal methods for reasoning about programming languages with user-centered design methods is a practical approach to designing languages that make programmers more effective. By doing so, designers can create safer languages that are more effective for programmers than existing languages. The thesis is substantiated by the introduction of PLIERS: Programming Language Iterative Evaluation and Refinement System. PLIERS is a process for designing programming languages that integrates formal methods with user-centered design.

The hypothesis that PLIERS is beneficial is supported by two language design projects. In these projects, I show how PLIERS benefits the programming language design process. Glacier is an extension to Java that enforces transitive class immutability, which is a much stronger property than that provided by languages that are in use today. Although there are hundreds of possible ways of restricting changes to state in programming languages, Glacier occupies a point in the design space that was justified by interview studies with professional software engineers. I evaluated Glacier in case studies, showing that it is expressive enough for some real-world applications. I also evaluated Glacier in lab studies and compared it to Java's `final` keyword, finding both that Glacier is more effective at expressing immutability than final and that Glacier detects bugs that users are likely to insert in code.

Blockchains are a distributed computing platform that aim to enable safe computation among users who do not necessarily trust each other. To improve safety relative to existing languages, in which programmers have repeatedly deployed software with serious bugs, I designed *Obsidian*, a new programming language for blockchain application development. From observations about typical blockchain applications, I derived two features that motivated the design of Obsidian. First, blockchain applications typically implement state machines, which support different operations in different states. Obsidian uses typestate, which lifts dynamic state into static types, to provide static guarantees regarding object state. Second, blockchain applications frequently manipulate resources, such as virtual currency. Obsidian provides a notion of ownership, which distinguishes one reference from all others. Obsidian supports resources via linear types, which ensure that owning references to resources are not accidentally lost.

The combination of resources and typestate results in a novel set of design problems for the type system; although each idea has been explored individually, combining them requires unifying different perspectives on linearity. Furthermore, no language with either one of these features has been designed in a user-centered way

or evaluated with users. Typical typestate systems have a complex set of permissions that provides safety properties, but these systems focused on expressiveness rather than on usability. Obsidian integrates typestate and resources in a novel way, resulting in a new type system design with a focus on simplicity and usability while retaining the desired safety properties. Obsidian is based on a core calculus I designed, Silica, for which I proved type soundness. In order to make Obsidian as usable as possible, I based its design on the results of formative studies with programmers. I evaluated Obsidian with two case studies, showing that Obsidian can be used to implement relevant programs. I also conducted a randomized controlled trial comparing Obsidian to Solidity, a popular language for writing smart contracts. I found that most of the Obsidian participants learned Obsidian and completed programming tasks after only a short training period; further, in one task, 70% of the participants who used Solidity accidentally inserted bugs that Obsidian's compiler would have detected. Finally, Obsidian participants completed significantly more tasks correctly than did Solidity participants.

Acknowledgments

First, I thank my advisors, Jonathan Aldrich and Brad Myers, and my unofficial advisor, Joshua Sunshine, who have nurtured my research career and consistently supported me in achieving my research goals. I am grateful for their technical guidance, their counsel in choosing research directions, and their mentorship.

Next, I am grateful for the amazing community at Carnegie Mellon, which has helped me flourish as a researcher and an educator. I have had wonderful conversations with Karl Crary, Andrew Faulring, Matt Fredrikson, Anna Gommerstadt, Claire Le Goues, Mor Harchol-Balter, Michael Hilton, Jan Hoffman, Eliezer Kanal, Christian Kästner, Amy Ko, Stefan Muller, Cyrus Omar, Bryan Parno, Andre Platzer, Frank Pfenning, Michael Sullivan, Mary Shaw. I am particularly grateful for Jenna Wise's help with the Obsidian formative studies. I also appreciate feedback from and discussions with James Noble and Alex Potanin.

Thank you to Gail Murphy for helpful advice as a member of my thesis committee and as part of the ICSE Doctoral Symposium in 2017, and to Frank Pfenning for being part of my thesis committee. I appreciate assistance from Werner Dietl, Michael Ernst, and Suzanne Millstein in understanding the details of IGJ. I also appreciate the help of the team at ZKoss.

I am grateful for the help of my numerous experiment participants, who will remain anonymous.

Thanks go to Rick Hull, Jim Laredo, Petr Novotny, and Yunhui Zheng at IBM, who provided useful technical and real-world insight during the Obsidian project. Thank you also to David Gould and Georgi Panterov at the World Bank, with whom I worked on the insurance case study.

Undergraduate and masters students made significant contributions to this work, and I enjoyed working with them immensely: Gauri Agarwal, Ishan Bhargava, Yannick Bloem, Suzz Glennon, Sachi Sharma, and Bobby Zhang. I am particularly grateful for students who spent an entire summer working with me through the Research Experience for Undergraduates in Software Engineering program: Miles Baker, Celeste Barnaby, Tyler Etzel, Gauri Kambhatla, Paulette Koronkevich, Whitney Nelson, and Reed Oei. I have learned much working with all of these students.

I am grateful to all of my colleagues in the iWork group at Apple, who taught me so much about how to write great software.

Finally, special thanks to my wife, Lauren, and my daughters Hannah and Rebecca, for the constant encouragement and support, as well as for facilitating all of the travel that is required to participate in the international research community. Likewise to my parents, Deborah and Robert, who nurtured my fascination with computers since I was a young child; to my brother, Joshua, for his support; and to my aunt Norma, who introduced me to computing at a young age at a time when very few people had personal computers (1985).

Contents

1 Introduction **1**
- 1.1 Thesis Statement . . . 3
- 1.2 Glacier . . . 3
- 1.3 Obsidian . . . 4
- 1.4 Technical contributions . . . 5
  - 1.4.1 Immutability . . . 5
  - 1.4.2 Glacier . . . 5
  - 1.4.3 Obsidian . . . 6
- 1.5 Methodological Contributions . . . 7
- 1.6 Thesis Outline . . . 9

2 Related Work **11**
- 2.1 Human-centered programming language design . . . 11
- 2.2 Design Methods . . . 14
  - 2.2.1 Methods for Requirements and Creation . . . 14
  - 2.2.2 Methods for Evaluation . . . 17
- 2.3 Immutability Systems . . . 19
- 2.4 Blockchain Programming Languages . . . 20
  - 2.4.1 Smart contract languages . . . 20
- 2.5 Aliasing, Permissions, and Linearity . . . 21
- 2.6 Typestate . . . 22

3 PLIERS: A User-Centered Process for Programming Language Design **23**
- 3.1 Introduction . . . 23
- 3.2 PLIERS . . . 27
  - 3.2.1 Evaluating PLIERS . . . 31

4 Exploring Immutability Features **33**
- 4.1 Introduction . . . 33
- 4.2 Overview of Concepts . . . 34
  - 4.2.1 Type of Restriction . . . 34
  - 4.2.2 Scope . . . 35
  - 4.2.3 Transitivity . . . 35
  - 4.2.4 Initialization . . . 36

4.2.5 Abstract vs. Concrete State . . . 36
4.2.6 Backward Compatibility . . . 36
4.2.7 Enforcement . . . 37
4.2.8 Polymorphism . . . 37
4.3 A Survey of Existing Systems . . . 37
4.3.1 Historical and Research Systems . . . 39
4.3.2 Popular Languages and Libraries . . . 41
4.3.3 Empirical Evaluations . . . 42
4.4 Usability of Features . . . 43
4.5 Interviews with Developers . . . 45
4.5.1 Methodology . . . 45
4.5.2 Results . . . 47
4.5.3 Implications on language design . . . 49
4.5.4 Limitations . . . 49
4.6 A Pilot Study of Transitive Immutability Restrictions . . . 50
4.6.1 Transitivity . . . 50
4.6.2 Study design and pilot results . . . 51
4.7 Conclusion . . . 53

5 Glacier: Transitive Class Immutability for Java 55
5.1 Introduction . . . 55
5.2 The Design of Glacier . . . 57
5.2.1 Evidence-based design . . . 57
5.2.2 Syntax and context . . . 57
5.2.3 Class immutability . . . 59
5.2.4 Restrictions of immutability . . . 60
5.2.5 Additional annotations for arrays . . . 61
5.2.6 Typecasts . . . 62
5.2.7 Type parameters . . . 62
5.2.8 Robustness to future changes . . . 62
5.2.9 Glacier formalization . . . 63
5.3 Evaluation: case studies . . . 63
5.3.1 Objectives . . . 63
5.3.2 Case study: ZK Spreadsheet Model . . . 65
5.3.3 Case study: Guava ImmutableList . . . 65
5.4 User study . . . 66
5.4.1 Methodology . . . 67
5.4.2 Participants . . . 69
5.4.3 Results . . . 69
5.4.4 Discussion . . . 71
5.4.5 Limitations . . . 72
5.5 Conclusion . . . 73

6 Introduction to Obsidian **75**
6.1 Requirements for Smart Contract Languages . . . 76
6.1.1 Strong static safety . . . 76
6.1.2 User-centered design . . . 77
6.1.3 Blockchain-agnosticism . . . 77
6.2 Obsidian Language Summary . . . 78

7 Permission Taxonomy and Permission System Design **87**
7.1 Permission Taxonomy . . . 87
7.2 Decomposition of permissions for Obsidian . . . 89

8 Formative Design in Obsidian **93**
8.1 Introduction . . . 93
8.2 Formative studies for Obsidian . . . 95
8.2.1 Basic design of typestate . . . 95
8.2.2 Fields in states . . . 98
8.2.3 Permissions: a qualitative study . . . 101
8.2.4 Comparing typestate and ownership approaches . . . 104
8.2.5 Threats to validity . . . 107
8.3 First usability study of Obsidian . . . 107
8.3.1 Participants . . . 108
8.3.2 Procedure . . . 108
8.3.3 Results and Discussion . . . 109
8.4 Conclusion . . . 110

9 Detailed Design and Formal Aspects of Obsidian **111**
9.1 Introduction . . . 111
9.2 Obsidian language design . . . 112
9.2.1 Type declarations, annotations, and static assertions . . . 112
9.2.2 State transitions . . . 112
9.2.3 Transaction scope . . . 113
9.2.4 Field type consistency and private transactions . . . 113
9.2.5 Dynamic State Checks . . . 115
9.2.6 Parametric Polymorphism . . . 117
9.3 System design and implementation . . . 118
9.3.1 Storage in the ledger . . . 118
9.3.2 Obsidian client programs . . . 119
9.3.3 Ensuring safety with untrusted clients . . . 121
9.4 Silica . . . 122
9.4.1 Silica Static Semantics . . . 124
9.4.2 Subtyping . . . 132
9.4.3 Silica Dynamic Semantics . . . 133
9.4.4 Silica Soundness and Asset Retention . . . 138
9.5 Obsidian Language Definition . . . 141

9.5.1 Obsidian syntax . . . 143
9.6 Translation from Obsidian to Silica . . . 144
9.7 Conclusion . . . 145

10 Obsidian Case Study Evaluation **147**
10.1 Case study 1: Parametric Insurance . . . 148
10.1.1 Motivation . . . 148
10.1.2 Requirements . . . 148
10.1.3 Design . . . 149
10.1.4 Results . . . 149
10.2 Case study 2: Shipping . . . 151
10.2.1 Motivation . . . 151
10.2.2 Results . . . 152
10.3 Case Study Summary . . . 155

11 An Empirical Study of Ownership, Typestate, and Assets in Obsidian **157**
11.1 Introduction . . . 157
11.2 The Solidity Language . . . 159
11.3 Study Design . . . 160
11.4 Participants . . . 162
11.5 Training . . . 163
11.6 Task Selection . . . 164
11.7 Auction Task . . . 165
11.7.1 Auction Results and Discussion . . . 166
11.8 Prescription Task . . . 168
11.8.1 Prescription Results . . . 169
11.8.2 Prescription Discussion . . . 170
11.9 Casino Task . . . 171
11.9.1 Casino Results and Discussion . . . 173
11.10 Post-study Survey Results . . . 176
11.11 Summary of Results . . . 177
11.12 Limitations and threats to validity . . . 179
11.13 Implications on PLIERS . . . 180
11.14 Conclusion . . . 180

12 Study Design Challenges and Solutions **183**
12.1 Training . . . 183
12.2 Recruiting . . . 185
12.3 High prototyping cost . . . 187
12.4 Interdependence of features . . . 187
12.5 High variance and external validity . . . 188
12.6 Time management . . . 189
12.7 Bias toward familiar languages . . . 190
12.8 Unsound proposals by participants . . . 190

13 Future Work and Conclusion **193**
13.1 Future Work . . . 193
13.1.1 Methods . . . 194
13.1.2 Applications . . . 195
13.1.3 Glacier . . . 196
13.1.4 Obsidian . . . 196
13.2 Conclusion . . . 197

A Auxiliary Judgements for Silica **201**
A.1 Program structure . . . 201
A.2 Reasoning about types . . . 204
A.3 Soundness Theorems . . . 209
A.3.1 Supporting Lemmas . . . 226

B Java Prescreen (used in user studies for Obsidian) **233**

C Case Studies of Obsidian **237**
C.1 Shipping . . . 237
C.2 Insurance . . . 248

Bibliography **265**

List of Figures

3.1 The phases of the PLIERS process, showing activities conducted in each phase. Designers can return to previous phases if evaluation identifies opportunities for improvement. . . . 28

4.1 Class diagram of starter code . . . 51

5.1 Alternatives we considered for immutability systems. . . . 58
5.2 Syntax of Glacier; gray boxes show differences with Featherweight Java. . . . 63
5.3 Formalization of Glacier semantics. . . . 64

6.1 Some common aliasing scenarios. (a) shows an object with one owner; (b) shows a shared object. . . . 80
6.2 A tiny vending machine that shows key features of Obsidian. . . . 82

8.1 Two of the options given to participants in the *basic design* study. . . . 97
8.2 Although participants preferred to have transactions nested inside state declarations (the first alternative), this desire conflicted with the need for transactions only reference fields that were in lexical scope. . . . 99
8.3 Variable declaration approaches . . . 105
8.4 An example question assessing understanding of ownership transfer. The correct answer is selected, since assignment transfers ownership. . . . 106

9.1 Obsidian's approach for handling transitions. . . . 113
9.2 An alternative approach for handling transitions. . . . 114
9.3 A dynamic state check example. . . . 116
9.4 Obsidian system architecture . . . 119
9.5 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 Fig. 6.2) to have Shared receivers, since top-level objects are never owned by clients. . . . 121
9.6 Abstract syntax of Silica. . . . 123

10.1 Invocations sent and results returned in a typical successful bid/claim scenario. . . . 150
10.2 Comparison between Obsidian and Solidity implementations of a Policy contract from the insurance case study. . . . 152
10.3 Initial design of the Shipping application (which does not compile). . . . 154

10.4 Revised design of the Shipping application. 154

11.1 An example practice question on assets from the Obsidian tutorial (showing the correct answer). 158
11.2 A side-by-side example comparing Obsidian code to Solidity code, taken from the two conditions of the Auction task (with minor changes to fit on this page). Code highlighted in yellow represents a correct solution; the rest was given to participants as starter code. In the Obsidian code, line 33 transfers ownership of the object referenced by `maxBid` to the `receivePayment` parameter. The new type of `maxBid` from then until line 35 is `Money@Unowned`. Line 35 re-establishes ownership in `maxBid` by transferring ownership from `money` to `maxBid`. 161
11.3 An example from the web-based Obsidian tutorial. 164
11.4 Sequence diagram given to participants to show what operations the Casino contract should support. 172

12.1 One question from the Obsidian tutorial (another is shown in Figure 11.1). The question assesses whether the participant has understood that at the ends of transactions, fields must be have types that match their declarations, and that returning a variable consumes any ownership in the variable. If a participant submits an incorrect answer, the survey tool informs them of their error so they can fix their misunderstanding. 184

B.1 Two design alternatives. 235

List of Tables

2.1 Summary of methods. Phrases are abbreviated: Requirements, Creation, Evaluation. 15

3.1 How PLIERS addresses common challenges in running user studies on programming languages. . . . 26
3.2 A summary of user-centered methods that we have found useful for studies in PLIERS. . . . 30

4.1 Summary of Dimensions . . . 38
4.2 Summary of Some Existing Systems (abbreviations are from Table 4.1) . . . 38

5.1 Summary of user study results . . . 69
5.2 Errors made by participants using `final` for immutability that remained after revision. Errors consist of failures to follow the advice in *Effective Java* [23]. . . 70

6.1 A summary of modes in Obsidian. . . . 81

7.1 Permission combinations when sharing is a property of references. Gray backgrounds show unsound combinations. Numbers index into Table 7.2, which shows the properties of each numbered combination. . . . 91
7.2 Permission semantics of useful combinations. Numbers are references from Table 7.1. . . . 91
7.3 Obsidian permission system . . . 92

8.1 Summary of all Obsidian user studies described in this paper and their participants. Participant P13 was in pilot studies. Participant numbering is consistent with prior distributed drafts, e.g., [45]. . . . 96
8.2 Usability test results. * indicates insufficient time to finish the task. N/A indicates insufficient time to start the task. . . . 109

9.1 Differences between Obsidian and Silica . . . 142

11.1 Participant experience (self-reported). . . . 163
11.2 Training times in Solidity and Obsidian conditions. . . . 164
11.3 Alignment between task design criteria and the tasks we designed. . . . 165
11.4 Auction task results. N=10 in each condition. . . . 166
11.5 Errors in Auction task. N=10 in each condition. . . . 167

11.6 Summary of Prescription task results. Times are shown as mean (standard deviation). N=10 in each condition. Two Solidity participants tried both static and dynamic approaches, and one Solidity participant made no changes, resulting in 11 Solidity attempts. 169
11.7 Summary of Casino task completions. 174
11.8 Summary of Casino task results among completed programs that compiled, showing correct solution rates among errors made by more than one participant. 174
11.9 Perceptions of ownership, states, and assets on a 1–5 scale (5 is best). Cells show average (standard deviation). * indicates that a Mann-Whitney U test shows a significant difference at $p < 0.05$. 181
11.10 Numbers of participants completing different numbers of tasks correctly in the two conditions. 181
11.11 Ranges of all (whether correct or not) solution lengths in lines of code. 181

Chapter 1

Introduction

Programming languages are *interfaces* that empower people to be more effective at creating software to achieve their goals than if they programmed in the language supported by hardware. They interface between the thought processes of a user, who is trying to create software, and the semantics of the machine on which the program will run. Programming languages form a *medium of expression* for creating software systems. In this thesis, I take the position that the primary goal of a programming language is to mediate between the needs of people and the needs of formal systems, such as abstract machines or real hardware.

Programming languages exist primarily to make programmers more effective at achieving their goals. Though much is known about how to analyze the theoretical properties of programming languages, this research focuses on how to design languages and analyze the impact of design decisions on programmers. When language designers make decisions about design questions that may meaningfully impact programmers, there is little scientific evidence — either grounded in cognitive science or in empirical data regarding programmer performance — on which to base their decisions. The lack of empirical evidence has been described in terms of the *language wars* [188]; Stefik and Hanenberg argue that without empirical data regarding programmer performance, there is no way to know which of several options is best for users, even when assuming a particular kind of programmer working in a known kind of project setting.

In some contexts, it would seem that most reasonable user needs can be fulfilled. Rarely, for example, does a web developer conclude that it is impossible to implement a desired feature (even though the implementation costs may be high). Programming languages, however, are different. Many problems that one might like a compiler to solve are undecidable [115, 170], and there is constant tension between providing strong guarantees and providing a simple language that enables users to understand the limits and properties of their tools. It is this tension that makes programming language design particularly interesting from a user-centered perspective: how can we design languages that have the properties we want, but in which people can be as effective as possible? Which properties are worth providing, given the complexity costs of providing them? After all, programming languages are *formal systems* in addition to being tools for users, and as such, it is beneficial for them to have strong, provable properties.

In this dissertation, I show methods I have developed to create and iterate on programming languages based on empirical data from users. By adapting *formative* methods from HCI, I collected data on which language design choices would be likely to be effective for users. By

adapting *summative* methods, I compared the resulting languages to existing languages and showed in what ways the new languages were more effective. By integrating techniques from the theory of programming languages, I obtained strong safety guarantees. I sought languages that were *principled*: which leveraged the theoretical principles of programming language design that resulted in languages that have provable properties.

In the service of developing new language design methods, I designed and implemented two programming languages. The first, *Glacier*, is an extension to Java for expressing and enforcing immutability. I used interviews with experts to generate hypotheses regarding which kinds of immutability would benefit software engineers; then I gathered qualitative data on use of an early prototype of my immutability specification system. After appropriate improvements motivated by those studies, I evaluated the final system in case studies and user studies to show that it is applicable to real-world systems and that users can use it more effectively than standard Java.

I also designed and implemented *Obsidian*, a new programming language intended for developing blockchain programs (sometimes called *smart contracts* [199]). Blockchain programming is a particularly compelling context in which to study language design. Although several languages are in use, including Solidity [73] and Go [64], bugs in Solidity programs have been exploited by hackers to steal money [81, 185]. Furthermore, blockchains are being proposed for use with high-stakes applications, such as banking, in which incorrect behavior can have serious consequences. Blockchains are designed to operate in situations where the users do not necessarily trust each other, but instead rely on the correctness of the programs; programs that do not behave as the users expect defeat the point of the platform.

In the design of Obsidian, I took the view that the language should provide safety guarantees that are relevant and useful for programs that are appropriate on the blockchain platforms. To reduce the training and high cost relative to formal verification, Obsidian uses type systems, which are a well-understood approach for helping users catch large classes of bugs at compile time rather than requiring them to catch these bugs through testing. In the future, a formal verification system could be layered on top of Obsidian to allow programmers to verify domain-specific properties.

Although Obsidian detects many bugs statically, Obsidian sometimes trades early bug detection for usability by moving some checks to runtime in order to improve flexibility and make the system more practical for programmers. For example, where the programmer inserts a dynamic state test, the Obsidian compiler inserts code to check to make sure any specifications on any owning reference are maintained (or the program is terminated). By automatically inserting dynamic state checks, Obsidian can allow more flexible aliasing configurations while maintaining soundness; this makes more user designs pass the type checker compared to a more restrictive approach. Obsidian only moves checks from compile time to runtime when this provides substantial usability advantages.

Validation of methods that are intended for expert language designers to use over the course of a long design process is challenging. One could imagine a study that recruited expert language designers, gave them a set of language design projects, and taught some participants the methods in this thesis. Then, one might try to run summative evaluations on the languages that they created. Unfortunately, this approach is infeasible; designing and implementing programming languages is typically an expensive, years-long effort with far too many independent variables to hope to be able to control variance enough to get a significant result (indeed, the design, implementation and evaluation of Obsidian itself has taken nearly four years). Rather than validate the methods in this

way, the thesis will show how they have been applied to obtain useful language design insights in specific cases — for Glacier and for Obsidian — and will show that these particular languages are measurably better than existing languages. I hope that in the future, others will adopt these methods and show how they are helpful in other use cases, as well as reveal limitations to their applicability that cannot be identified in a small number of case studies.

In this introduction, Related Work, and in the conclusion, I use the singular first person, but in other chapters I use *we* to acknowledge collaboration with others.

1.1 Thesis Statement

Combining theoretical and human-centered methods in the design and evaluation of programming languages can lead to programming languages that make it easier for users to write demonstrably safer, more-correct programs than if they wrote those programs in existing programming languages.

The thesis is supported by a new process, *PLIERS*: **P**rogramming **L**anguage **I**terative **E**valuation and **R**efinement **S**ystem, for language design. PLIERS shows how theoretical and human-centered methods can be combined into an integrated language design process. I developed and evaluated PLIERS in the context of two language design projects, in which I show how using PLIERS can benefit language designers.

1.2 Glacier

Glacier is an extension of Java that enforces transitive class immutability. I first studied the existing space of immutability restrictions and found hundreds of different possible kinds of mutation restrictions. I then conducted a study of expert software engineers and used their suggestions to form a hypothesis: transitive class immutability, if provided by a programming language, would express a commonly-used form of immutability that would be helpful in preventing bugs. Based on this, I designed and implemented Glacier, which adds the `@Immutable` annotation to Java and enforces transitive class immutability on `@Immutable`-annotated classes at compile time.

I evaluated Glacier with a two-part user study, which compared Glacier to Java `final`. I randomly assigned participants to use either Glacier or `final`. First, I asked them to specify immutability in a very small codebase with their given language. I found that although 90% of the Glacier users were able to do so effectively, all of the `final` users made mistakes attempting to specify immutability, a statistically significant difference. Next, I asked participants to complete two programming tasks in a codebase in which immutability was already specified. In each task, 70% of the Glacier users succeeded, but more than half of the `final` users accidentally modified immutable structures even though almost all of them believed they had completed the tasks successfully.

Although Glacier represents a small part of a complete language design, it shows that using human-centered design approaches can be effective in developing language constructs that help programmers avoid writing buggy code while facilitating writing correct code. In particular,

Glacier provides strong safety guarantees that I was able to show using a lab study can be leveraged effectively by programmers.

In addition to a lab study of Glacier, I also conducted a case study in which I used Glacier to specify immutability in two real-world codebases: a spreadsheet implementation and an immutable collections library. This showed that Glacier is expressive enough to reflect real-world use cases.

1.3 Obsidian

To show how user-centered design methods can be useful in the creation of new programming languages, I created Obsidian, a new programming language for blockchain application development. Blockchains are a distributed computing platform that aim to enable safe computation among users who do not necessarily trust each other. Blockchain software carries particularly high stakes due to these properties:

1. Immutability: blockchain transactions cannot be reversed, and on some blockchain platforms, programs cannot be directly modified after deployment. This means that bugs can be impossible to fix once deployed.
2. Applications: many of the proposed blockchain applications govern important transactions, many of which govern resources such as money. Bugs or vulnerabilities in these programs can result in loss or theft of money or other resources. Since blockchains are intended to facilitate trust, it is crucial that the programs be correct so that users can rely on them.

The main objective of Obsidian is to provide relevant, static correctness guarantees in a way that enables programmers to be more effective at writing safe programs. Rather than providing guarantees that programs meet arbitrary specifications via a general-purpose verification mechanism, which is so far too difficult and expensive to be practical for most programmers, Obsidian is grounded in two observations:

1. Many blockchain programs are stateful at a high level: they implement state machines in which the operations that are permitted depend on the state of the program. Existing blockchain programming languages do not protect statically against calling transactions that may be unavailable due to the current state of the program, instead requiring runtime checks of state.
2. Many blockchain programs manipulate resources, such as virtual currency or votes in organizations. These resources must be tracked correctly, but existing blockchain programming languages do not facilitate static reasoning about correct resource usage. In particular, resources should never be accidentally lost.

Obsidian leverages these observations by incorporating *typestate* [5] and *linear types* [214] to provide strong static guarantees that are relevant to blockchain programs. The goal is to use human-centered design and evaluation methods to make it practical for programmers to obtain these guarantees while still completing development efficiently and effectively. In addition, the design of Obsidian is itself a novel combination of typestate and resources. The space of permissions systems for managing this combination is particularly complicated; Obsidian shows how to combine these ideas in a coherent, understandable way.

In addition to informing the design with user-centered methods, I evaluated Obsidian in a summative usability study (§ 8.3) and a randomized controlled trial (RCT) comparing it to Solidity (chapter 11). Most of the Obsidian participants were able to use ownership effectively, and most of the Solidity participants accidentally inserted bugs that the Obsidian type system can detect. However, in a task that was more open-ended than the others, Obsidian participants spent longer on their work than the Solidity participants and abused features of the type system to accidentally lose assets. This corresponds with a traditional assumption that statically-typed languages may have additional costs in prototype development, and suggests that more training of users or future design research may be needed to help programmers use strong type systems effectively.

with two case studies

1.4 Technical contributions

1.4.1 Immutability

The space of immutability systems is complex. For example, if an object is immutable, does that mean only that its fields cannot be reassigned, or that its fields must only refer to immutable data? I used interviews with software engineers to identify a promising subspace within the large space of immutability systems. I also created a prototype, IGJ-T, and identified usability problems that I addressed in Glacier. Contributions include:

1. After reviewing the existing literature and implemented systems in this area, I developed a classification system for mutability restrictions in programming languages, showed where some existing systems fit in this classification, and discussed possible usability implications of existing design choices.
2. I interviewed eight expert software engineers to identify hypotheses regarding existing language features and tools, and what the perceived needs of software engineers are. I showed that existing programming language features and research tools do not adequately address the issues encountered by practitioners. I extracted design recommendations from the results of the interviews.
3. I created IGJ-T, an extension to IGJ that enforces *transitive immutability*, which addresses problems that our interview participants described. I iteratively evaluated IGJ-T with three pilot users and refined the study based on the feedback.

1.4.2 Glacier

Glacier's design shows it is possible to obtain both useful expressiveness and simplicity in one design for an immutability system. Glacier provides strong, useful immutability properties that users can actually use effectively. Contributions include:

1. A definition and formal model of transitive class immutability as an extension to Featherweight Java [100];
2. An implementation of that model in a tool called *Glacier*, which enforces transitive class immutability in Java. By enforcing only the kind of immutability for which there is the

strongest empirical support, I have achieved significant simplifications relative to existing type system approaches.

3. Evaluations of Glacier in two case studies on real software projects that showed that Glacier captures a kind of immutability appropriate for those projects.
4. The first formal user study of any immutability system. We compared Glacier to `final` and found that all ten participants who used `final` wrote code that had bugs or security vulnerabilities, even after having been trained on correct `final` usage, in a situation in which Glacier statically detects those problems. Almost all the Glacier users were able to complete the tasks successfully.

1.4.3 Obsidian

Obsidian reflects a novel set of language design features that have previously never been integrated: an object-oriented language that supports both typestate and linear resources. The type system provides strong safety guarantees for practical programs while remaining accessible for many programmers. The design of the type system is particularly interesting in the presence of aliasing; compromising between expressiveness and simplicity enables many important properties to be verified by the compiler but also enables easy understanding. Obsidian is based on a core formalism, Silica, for which I proved type soundness. I conducted two case studies showing that Obsidian can be used to implement relevant blockchain programs. I also conducted a user study showing that participants were able to use Obsidian to solve programming problems, and that participants using Solidity (a popular language for blockchain programming) frequently inserted bugs that Obsidian's compiler detects.

Contributions pertaining to the formal design of Silica and Obsidian include:

1. I 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. I 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. I describe Silica, the core calculus that underlies Obsidian. I 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 I am aware) that supports assets.

Contributions pertaining to the case study evaluation of Obsidian include:

4. The aliasing structure in the blockchain applications I 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.
5. 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.

6. I and my collaborators were able to successfully build nontrivial smart contracts in Obsidian that reflected real-world use cases.

Contributions pertaining to the summative usability evaluation include:

7. Showing that Solidity participants (the control condition) in a lab setting frequently insert bugs that lose assets in a small programming task that is representative of a real smart contract programming problem.
8. Showing that most of the Obsidian participants in my study were able to use ownership to solve a security problem, which participants had great difficulty with in earlier versions of Obsidian.
9. Showing that although the participants were able to use ownership effectively, they were less successful in using typestate to obtain static safety guarantees, and further work may be required in this area.

1.5 Methodological Contributions

In addition to the concrete designs of Glacier and Obsidian, the thesis shows how methods from human-computer interaction can be adapted for effective use in programming language design. In particular, methods can be used in a way that gathers both formative data for hypothesis generation as well as summative data for hypothesis evaluation. Integrating these methods with a theoretical perspective that enables designing a language with well-defined properties yields PLIERS, which is described in detail in chapter 3. Rather than relying only on one's own insight about programming and programmers, using human-centered methods enables the designer to ground the design in the needs and performance of users. Furthermore, these methods can be used in conjunction with traditional formal methods to arrive at a *sound* language that provides needed safety properties.

One example method is *back-porting language design decisions* into a familiar language to enable study of a particular language design decision in isolation. If participants were required to learn an entire new language and were confused during tasks, it might be difficult to identify which aspects of the design were confusing. By isolating aspects of interest, one can study design decisions with fewer confounds. Like other methods, of course, this method has limitations. The design decisions of interest may not be easily portable to a language in which one can find participants, and furthermore, individual decisions may not be orthogonal. Nonetheless, because features are typically designed to be as orthogonal as possible, I have found this approach to lead to useful insights into user behavior and expectations (chapter 8).

The approach of designing a programming language based in part on user experiments is fraught with confounds. Is it not likely that the best programming language for participants in a short-term study is one that the participants already know? When evaluating language constructs, one might expect that the constructs that seem most familiar may appear to be the best in a short-term study even if this not the case long-term. Certainly, the results of a user study of a particular language design will depend on the participants, including the kinds of programming

with which they are most familiar (object-oriented, functional, etc.), their level of experience and skill, and the particular languages they are skilled in. Any attempt to evaluate language designs with people must address these confounds.

I address the problem of confounds with several techniques. First, showing benefit of a language feature that prevents or detects bugs does not require that it be faster – only that it can be used effectively – since preventing bugs may be worth some amount of development cost. Of course, each study must define "effectively." In my studies, I aimed to show that my tools resulted in statistically fewer bugs (in a given class) than an existing, standard tool, and at the same time saw no statistically significant difference in task times. Of course, the task times may be different by an amount that was not statistically significant with the sample size I used. In other words, I conducted experiments that focused on one particular research question of importance, rather than trying to show that the tools I developed were better in all ways.

Second, task completion times and rates are most meaningful after an appropriate training period has elapsed. This restricts the analysis to languages that can be taught effectively in a short-enough time for a study. If a language requires more training, perhaps the recruitment criteria can be adjusted to find participants who need less training time. Another approach is to focus the study on a smaller portion of the language – one that *can* be taught in a short period of time. In other words, the techniques I developed can show that some kinds of languages represent improvements over existing ones, but the techniques do not need to be applicable in all possible language design and evaluation scenarios in order to be useful in others.

Third, I triangulate among many different methods. For example, I used interviews to assess the extent to which programmers believed an approach might be helpful and the extent to which they encountered particular problems. Then, I did qualitative studies to obtain in-depth details about the usability of particular designs, and finally I did quantitative studies to try to show statistically significant benefits. If, as I found, the methods give consistent results, that provides stronger evidence in support of the design. Inconsistent results among methods may serve to identify useful hypotheses regarding explanations for the discrepancy, which can be evaluated and addressed. Perhaps the users requested features that they would not, in fact benefit from, or the problems the users face are different from the problems the users perceive. Perhaps a feature similar to what the users requested might be effective, but usability problems would need to be addressed first. Or perhaps although the benefit has not yet been quantified successfully due to variance or some other problem, qualitative evidence of benefit suffices in certain cases to motivate future development or deployment.

In summary:

1. I show by example how I have adapted several formative study techniques, such as natural programming, Wizard of Oz, rapid prototyping, cognitive dimensions of notations analysis, and interview studies to inform the design of Glacier and Obsidian. I informed the results of these techniques with implications of the theory of programming languages to develop languages that were effective for users and also achieved our safety objectives. I found that the adapted methods were effective when used in the context of a design process, PLIERS.
2. I show how I conducted summative usability studies and randomized controlled trials on new programming languages. By developing ways of teaching the languages efficiently, effectively, and consistently, I was able to conduct usability studies of programmers using

novel programming languages.

3. I integrate the above study techniques with a theoretically-grounded perspective on programming language design into PLIERS.

1.6 Thesis Outline

Chapter 2 describes related work. In chapter 3, I describe PLIERS, which explains how to integrate user-centered and theoretical methods together to design programming languages. Then, the dissertation continues by showing how PLIERS was useful in the context of two language designs. First, chapter 4 describes an analysis of the space of immutability features in programming languages. It also reports on an interview study I conducted and a proposal for a specific design for expressing immutability based on the interviews. In chapter 5, I describe Glacier, the immutability language extension I developed, and a lab study showing its benefits.

The next part of the dissertation focuses on Obsidian. Chapter 6 introduces the requirements I identified for the Obsidian language and summarizes the final language with an example. Chapter 7 gives an analysis of the space of permissions systems, showing where Obsidian's design fits in the space. Chapter 8 describes the formative studies I conducted to inform the design of Obsidian; chapter 9 describes Obsidian formally and presents soundness theorems for Silica, a core formalism for Obsidian. Chapter 10 describes two case studies I conducted that evaluated Obsidian. In chapter 11, I describe the empirical study I conducted to compare Obsidian to Solidity.

In chapter 12, I describe challenges I faced when designing user studies of programming languages, and the techniques I have developed to overcome those challenges and obtain useful insights from user studies. I propose future work and conclude in chapter 13.

Chapter 2

Related Work

Newell and Card argued for the use of HCI methods in programming language design in 1985: "Millions for compilers but hardly a penny for understanding human programming language use [134]." Morrisett reiterated this problem in 2009, arguing that a programming language is a medium for communication among humans, but we lack principles for evaluating this aspect of languages [153]. In this chapter, I first summarize the history of empirical methods for design and evaluation of programming languages. Next, I describe approaches that other researchers have taken to develop user-centered design methods in other domains. Then, I summarize the related work for Glacier and Obsidian, including discussions of immutability and blockchain programming languages. Finally, I describe work that relates to the technical approaches we use in Obsidian, including aliasing, permissions, linear types, and typestate.

2.1 Human-centered programming language design

The Empirical Studies of Programmers workshops from the 1980s and 1990s focused in large part on a cognitive science approach to studying programmers: can we build models of cognition that explain programmer behavior? Key results include describing techniques used by programmers when working with code, such as identifying key lines (beacons), relating program details to the problem domain, and using both top-down and bottom-up understanding techniques. The ESP literature provided insights into the problems that people have when using existing languages.

Some of the work in ESP workshops studied professional programmers. For example, Pennington used theories of understanding of natural language text to model expert programmers' comprehension of programs [149], finding that procedural knowledge (rather than knowledge of functional units) dominated their understanding. The study was conducted on COBOL programs, which were likely structured substantially differently from modern software. Furthermore, no libraries or frameworks were used, so the fact that the programmers could see and consider all relevant code may have resulted in a substantially different kind of programming task than the ones that we consider today. However, the approach suggests that a cognitive modeling approach may help derive a theory of programmers that is relevant for designing programming languages. In this work, I rely on more recent, heuristic-based approaches, such as the Cognitive Dimensions of Notations [85], which are applicable in more general domains, and which came out of the

cognitive science-based approach that was central in the ESP work.

Visser described a four-week observational study of one professional programmer working in a declarative, domain-specific language [212]. Visser noted that obtaining mental models was challenging because the programmer found think-aloud very difficult while working on problems, but made observations about the structure of the programmer's work. For example, the programmer used analogical reasoning and examples to help reason about the software; used both top-down and bottom-up work styles; and sought consistency in the program. This work provides an empirical foundation for some requirements of programming environments, such as allowing creation of interfaces separately from implementations, and providing tools to standardize notation (e.g. style linting tools).

Vans et al. conducted study of the comprehension process and information needs of programmers in industry doing maintenance tasks [208]. Some of the understanding techniques that the programmers used were similar to methods that were observed in novices as well, including top-down, bottom-up, and code-tracing methods, but the professionals used a much wider variety of techniques than had been observed in novices, such as generating and abandoning large numbers of hypotheses regarding the programs. This suggests that programming language design studies conducted with students can give some guidance regarding languages intended for professionals, but such studies may be limited in the kinds of techniques that the participants use. In many of the studies presented in this work, I recruited experienced students, who in many cases had several years of professional programming experience. This approach allowed me to broaden the set of techniques my participants would use to accomplish their tasks and make my results more generalizable relative to using only novice programmers.

Guindon et al. used protocol analysis [70] to analyze think-aloud protocols from three experienced programmers who were asked to design software to solve an elevator-scheduling problem [86]. Guindon et al. observed breakdowns in process that arose from lack of knowledge (e.g., of the problem domain) and from cognitive limitations (e.g., capacity of short-term memory). Because this work consisted of a think-aloud study of programmers, it shares several threats to validity with the qualitative work I did: a task that may not reflect real-world tasks; short duration of the task, which was concentrated in a lab-based two-hour session; and a sample of programmers that may not be representative of programmers in general. My approaches to mitigating external validity were similar to theirs. I recruited from students who were likely to have some professional experience and do not claim that all programmers will encounter the same difficulties that they did. I do not claim that I observed all possible problems that users might have when using the tools I gave them. Instead, I argue that addressing the problems we observed is likely to help *some* relevant users be more effective in achieving their goals. In my approach to think-aloud studies, I conducted a form of protocol analysis by analyzing notes taken by the experimenter and screen recordings of the participants doing the tasks.

Direct observations of work and interviews have both been used to understand how teams work together to develop software. Walz et al. analyzed videos of teams conducting a requirements analysis to study conflict patterns [215]; Krasner et al. interviewed members of 19 software development teams to understand team communication [113]. Although my work focused on individual developers, I used multiple methods where appropriate. This work shares threats to external validity with other small-sample studies, including the ones I conducted. Krasner et al. mitigated these risks by choosing diverse teams to study. Although Walz et al. only studied one

team, they studied the team over 43 meetings over four months.

A substantial amount of prior work on the usability of programming languages focuses on novices. For example, HANDS [146], Helena [36], and Scratch [169] aimed to make it easier for novices to write programs. HANDS, in particular, introduced the *Natural Programming* technique, which I leveraged and adapted in this work. Stefik et al. also focused on novices, collecting quantitative data on their error rates [192]. Designing languages for novices is substantially different from designing languages for experienced programmers. For example, languages for novices typically focus on learnability. In contrast, languages for professionals commonly include additional complexity, in part resulting from the kinds of safety properties that are beneficial when building real systems.

Other work has focused on programming tools for end-user programmers, whose primary goal is not to write software but rather to accomplish goals in some particular domain [110]. For example, Blackwell and Burnett developed the Attention Investment model and applied it to a research spreadsheet tool, Forms/3 [21]. Peyton Jones et al. used Cognitive Dimensions [85] and Attention Investment to provide a new kind of user-defined functions in Excel [101]. My work is focused on methods that address the unique challenges of complexity that result from targeting professional programmers and software engineers.

High-quality error messages are a key component of a usable compiler. Some work in this area has produced guidelines for creating error messages that are effective for novices [12]. My work focuses on professionals, but the guidelines given in that work (for example, *show solutions or hints*) are relevant to languages for professionals as well.

RCTs (randomized controlled trials) have been used to compare different programming language designs. For example, Uesbeck et al. investigated the impact of lambdas in C++ [204], and Endrikat et al. looked at static typing [69]. That work is a useful complement to this work, but the focus here is on using low-cost, practical *qualitative* methods to inform the entire language design process. In contrast, quantitative summative studies require high-fidelity prototypes in order to obtain measurements that can be expected to generalize to the final system. These prototypes can be very expensive to build for complex programming languages.

Another approach to programming language evaluation is to compare designs via crowdsourcing methods. Chamberlain [35] compared functional-style to literal-style approaches for specifying topology of streaming applications (i.e., pipes-and-filters style applications) using Mechanical Turk, finding that users were more likely to prefer literal-style specifications, and experienced programmers were more likely to understand the literal-style specifications than the functional-style ones. Wilson et al. [217] investigated crowdsourcing more esoteric language design decisions, finding low consistency (people did not give consistent answers when asked similar questions repeatedly) and low consensus (people did not agree with each other on which design choice was best). Crowdsourcing approaches can scale well, but typically require that the studies be of relatively short duration. My work focuses on higher-bandwidth qualitative methods and on evaluation approaches for languages for professionals, and serves to complement crowdsourcing approaches.

The HCI literature includes many different language designs as well as other kinds of tools for programmers. For example, Dog/Jabberwocky [3], Protovis [25], Reactive Vega [176], and InterState [144] are all languages or APIs that make it easier for programmers to accomplish their goals. Those papers describe only the final designs of those systems and summative

usability studies. Here, I focus on methods that can be used *during* the design process and gives recommendations that are useful in preparing a summative evaluation.

Finally, there is a variety of methodological guidance in SE and HCI that is applicable to studies of programming languages. Ko et al. discussed techniques for doing empirical studies of tools for software engineers [111]. Buse et al. conducted a systematic literature review, observing increasing use of user evaluations in software engineering research [31]. Verner et al. gave guidelines for industrial case studies in software engineering research [211]. Perry et al. gave a tutorial on case study methodology for software engineers [151]. Likewise, Shneiderman and Plaisant gave recommendations for using case studies for information visualization tools [184].

2.2 Design Methods[1]

In this work, I adapted existing formative and summative methods for obtaining insight about user-facing systems. I divide the design work into two phases, between which the designer alternates in a cyclic fashion; here, I summarize existing work on methods in the two phases. Table 2.1 summarizes methods that can be used in user-centered programming language design.

1. In the *requirements elicitation and creation* phase, the designer studies the application domain for the language. The designer creates a draft version of the language, likely including a language specification and language implementation.
2. In the *evaluation* phase, the designer evaluates how well the language fulfills its requirements.

2.2.1 Methods for Requirements and Creation

Interviews can be a valuable source of information for areas in which researchers can find experts. These can be a useful approach to quickly obtain knowledge about existing problems and their existing solutions. For example, LaToza interviewed software engineers to develop hypotheses regarding how developers established and maintained mental models of software systems [117]. Interviews are limited in external validity because it may be difficult or impossible to interview a representative sample of any particular population. The results strongly depend on the participants themselves as well as the skill of the interviewer in eliciting as much useful information as possible with minimal bias.

Surveys are a useful way to assess opinions and experience among a large sample, for example for assessing whether a proposed problem is one that a large fraction of practitioners face, or assessing which problems are the most important to solve from a practitioner's point of view. Some researchers have also used surveys to get direct insight into programming language designs [203], but the results have been inconclusive regarding specific design guidance. Most surveys ask people what they believe, but in some cases people's beliefs do not lead to designs that benefit users in practice. Furthermore, survey results can be difficult to interpret or clouded with noise. Sometimes, little verifiable information is known about participants, and there may be motives

[1] Portions of this section previously appeared in [43].

Method	Phases	Key benefits	Challenges and limitations
Interviews	R, C	Gathers open-ended qualitative data from experts	Depends on skill of interviewer and selection of participants; results may not generalize
Surveys	R, C	Assesses opinions among a broad audience	Output is subjective; may not reflect reality
Corpus studies	R, C	Assesses incidence of problems or applicability of solutions in a large dataset	Depends on appropriate datasets and efficient methods of analysis
Natural programming	R, C	Obtains insights from people without biasing them toward preferred solutions	Data may be biased toward participants' prior experiences
Rapid prototyping	R, C	Facilitates efficient design space exploration	Lack of fidelity in prototypes may hide faults
Programming language theory	R, C, E	Ensures sound designs	High cost; formal work too early may limit ability to iterate, but formal work too late can waste time on unsound approaches
Software engineering theory	R, C, E	Improves design practicality	Unclear how to prioritize recommendations when they conflict
Qualitative user studies	R, C, E	High-bandwidth method to obtain insight on user behavior when using systems	Results may not generalize; Results depend on skills of experimenter and participants
Case studies	E	Tests applicability to real-world cases; allows in-depth explorations of real-world difficulties	Requires finding appropriate cases; generalizability may be limited
Expert evaluation	E	Benefit from experience acquired by experts	Biased by opinions of experts, which may not reflect real-world implications of the design
Performance evaluation	E	Reproducible way of assessing performance	Results depend heavily on selection of test suite
User experiments	E	Quantitative comparison of human performance across systems	Results may not generalize to non-trivial tasks, other kinds of users, long-term use, or use on large systems
Formalism and proof	R, C, E	Provides definitive evidence of safety	Results are limited to the specific theorems proven

Table 2.1: Summary of methods. Phrases are abbreviated: Requirements, Creation, Evaluation.

that detract from data validity (e.g., Mechanical Turk workers may want to complete the survey as fast as possible to maximize their hourly wage).

Corpus studies, which analyze existing corpora of code, can show the prevalence of particular patterns in existing code, including patterns of bugs in bug databases. For example, Callaú et al. [33] investigated the use of dynamic features in Smalltalk programs, Malayeri et al. [122] investigated whether programs might benefit from a language that supported structural subtyping, and Kery et al. studied how Java programmers used exception handling features [107]. Corpus studies can show that a particular problem occurs often enough that it might be worth addressing; they can also show how broadly a particular solution applies to real-world programs, as in Unkel and Lam's analysis of stationary fields in Java [206]. However, it can be difficult to obtain a representative corpus. For example, although GitHub contains many open source projects, they can be difficult to build; it can be difficult to sample in an unbiased way; and open source code may not be representative of closed source code.

Natural programming [132] is a technique to elicit how people express solutions to problems without any special training. It aims to find out how people might "naturally" write programs. These approaches have been useful for HANDS [146], a programming environment for children. However, the results are biased by participants' prior experience and education, and results depend on careful choice of prompts to avoid biasing the results.

Rapid prototyping is commonly used in many different areas of human-computer interaction research, and can be used for language design as well [131]. Low-fidelity prototypes, such as paper prototypes, can be used to obtain feedback from users on early-stage design ideas. For example, LaToza used paper protypes to refine Reacher, a tool for helping developers answer reachability questions about code [116]. Wizard-of-Oz testing involves an experimenter substituting for a missing or insufficient implementation [84]. However, low fidelity prototypes may differ in substantive ways from polished systems, misleading participants. The results depend on the skill and perspectives of the experimenter and the participants, which threatens validity.

Participatory design [24, 145] invites domain experts to help explore the design space and analyze tradeoffs. In the case of language design, the assumption is that the specific expertise is likely to complement the general language design expertise of the language designer. The use of participatory design arose as a theme in one symposium on end-user development [62]. For example, Kanstrup described the benefits of participatory design in creating support tools for people with diabetes [105]. Although many people without substantial programming language design expertise have designed programming languages for themselves, here we are interested in fusing the domain expertise of the users of the languages with the expertise of a programming language designer.

Programming language and software engineering theory provide a useful guide when considering the requirements for a programming language. For example, the guarantees that a transitive immutability system can provide in the areas of both security and concurrency—which have been well-established in the programming language theory literature—were key reasons that I chose transitive immutability for the Glacier type system (chapter 4). Similarly, an understanding of how modularity affects modifiability from the software engineering literature [148] motivates the module systems present in many languages, and more recent theories about how software architecture [183] influences software development motivated the design of the ArchJava language [4]. However, theoretical guarantees that pertain to optional language features will

not be obtained if the features are misunderstood or not used. Furthermore, guarantees can be compromised by bugs in unverified tool implementations.

2.2.2 Methods for Evaluation

In "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community," Stefik and Hanenberg argued for empirical evaluation of programming languages [188]. In a separate paper, Stefik at al. described a systematic literature review of PPIG, PLATEAU, and ESP, finding that those papers rarely report on the impact of language design on users [190]. This subsection describes approaches that designers and researchers have taken to obtain evidence regarding the effectiveness of their programming language designs.

Qualitative user studies have been used to evaluate many different kinds of tools, including programming languages [146], APIs [133], and development environments [112]. Some of these consist of *usability analyses*, in which participants are given tasks to complete with a set of tools and the experimenter collects data regarding obstacles the participants encounter while performing the tasks. Unlike randomized controlled trials, these are usually not comparative; that analysis is left to a future study. Instead, they focus on learning as much as possible in a short amount of time in order to *test feasibility* of a particular approach and *improve the tool* for a future iteration of the design process.

Qualitative user studies can also be used to understand a problem that a language design is intended to solve, and help to guide other research methods used to evaluate the eventual solution. Sunshine et al. studied programmers solving protocol-related programming problems that were gleaned from real StackOverflow questions in order to understand the barriers developers face when using stateful libraries [197]. The results of the study were useful in developing a language and its associated tools, and produced a set of tasks that were used in a later user experiment. Because of the qualitative user study, Sunshine et al. knew these tasks were the most time-consuming component of real-world programming problems, mitigating the most significant external threat to the validity of the user experiment.

Qualitative user studies are usually limited to short-duration tasks with participants that researchers can find. In practice, this sometimes limits the sizes of the programs that the tasks concern because larger programs typically require more sophisticated participants and more participant time. Although a typical qualitative user study might only take an hour or two per participant, even a small real-world programming task might take a day or more.

Case studies [221] show *expressiveness*: a solution to a particular programming problem can be expressed in the language in question. Many case studies aim to show *concision*, observing that the solution is expressible with a short program, particularly in comparison to the length of a typical solution in a comparison language. Case studies are particularly helpful when the language imposes restrictions that might cause a reader to wonder whether the restrictions prevent application of the language to real problems.

Case studies can also be used to learn about how a programming language design works in practice. For example, Aldrich et al. used exploratory case studies on ArchJava to learn about the strengths and limitations of the language design and to generate hypotheses about how the approach might affect the software engineering process [4].

Case studies have limited external validity because they necessarily only consider a small set of use cases (perhaps just one). As a result, the conclusions are biased by the selection of the cases. Furthermore, the results may not generalize to typical users, since the case studies may be done by expert users or even the creators of the system under evaluation. Verner et al. provided guidelines for industrial case studies in software engineering research [211]. Perry et al. gave a tutorial on case study methodology for software engineers at ICSE 2004 [151]. Likewise, Shneiderman and Plaisant gave recommendations for using case studies for information visualization tools [184].

Expert evaluation methods, such as Cognitive Dimensions of Notations [85] and Nielsen's Heuristic Analysis [138], provide a vocabulary for discussing design tradeoffs. Although they do not definitively specify what decision to make in any particular case because each option may trade off with another, they provide a validated mechanism for identifying advantages and disadvantages of various approaches. This approach has been widely used in the visual languages community. However, expert evaluation requires access to experts and a validated and relevant set of criteria. The traditional criteria, such as Cognitive Dimensions of Notations, have not yet been validated to show that applying the criteria tends to result in measurable improvement in programmer task performance.

Performance evaluation, typically via benchmarks, is well-accepted for comparing languages and tools. Performance evaluation can be critical if it is relevant to the claims made about a language, but many popular languages are not as fast as alternatives (consider Python vs. C), so it is important to decide how much performance is required. SIGPLAN released a checklist [13] for empirical evaluations of programming languages; although is title is "Empirical Evaluation Checklist," it describes only performance evaluations. The checklist describes limitations of this approach, such as a mismatch between benchmark suite and real-world applications, an insufficient number of trials, and unrealistic input.

User experiments, including randomized controlled trials (RCTs), have been used to address a variety of programming language design questions. Endrikat et al. found that static typing improves software maintainability [69]. Prechelt and Tichy investigated the benefits of type checking [157]. Hanenberg et al. compared static and dynamic type systems [90], finding benefits of static type systems in both documentation and compile-time checking. However, Uesbeck et al. found that although C++ lambdas are a highly-touted feature of C++, they actually impose costs to programmers [204]. Together, these results suggest that although static typechecking can be beneficial, it is important to do usability studies to assess the impact of any proposed language feature. Stefik et al. developed methodology to evaluate syntax choices for novice programmers [191].

In some ways, RCTs represent the gold standard for summative evaluations. However, they do not always lead to insights that can be used to design or improve systems, and unless they are supplemented by theory (e.g., gleaned from qualitative studies), it can be difficult to be certain that results on a narrow problem studied in the laboratory will apply to a more complex real-world setting. For example, Uesbeck et al. discuss in what contexts their conclusions about C++ lambdas might apply [204], but not how one might improve lambdas to retain possible advantages but mitigate identified shortcomings.

RCTs for complete, novel programming languages (as opposed to ones that were already familiar to the participants) are extremely rare. Quorum [191] and HANDS [146] were both evaluated in user studies. Compared to Hanenberg et al. [90], we focus more on an experimental

design that minimizes time required by participants and on reporting detailed data on errors made by participants. Other work has focused on language extensions, such as for software transactional memory [147], or APIs and libraries, such as Dog/Jabberwocky [3], Protovis [25], Reactive Vega [176], and InterState [144].

Techniques for doing empirical studies of tools for software engineers [111] are also relevant to studies of programmers. Buse et al. conducted a systematic literature review, observing increasing use of user evaluations in software engineering research [31] in general.

Formalism and proof are traditional tools for showing that a specific language design has particular properties, such as type soundness [152]. In many languages, a formal model provides key insight that inspires a new language design; in these cases, the formal analysis might be the *first* step in a language design. However, in other languages, a formalism serves primarily to provide a specification and a safety guarantee, in which case this work might be done much later.

A typechecker provides some safety guarantees once a program typechecks, but one must compare the difficulty of writing a type-correct program to the difficulty of obtaining safety some other way (for example, with runtime checks, at the cost of deferring verification to runtime) and to the option of not providing the guarantee at all. In some systems, safety guarantees are not necessary; for example, the consequences of a bug in a video game may be smaller than the consequences of a bug in avionics software. Recently, Misailovic et al. have argued that in many cases, approximating the language semantics suffices [128].

Formal verification via tools such as Dafny [118] or Coq [15] can provide even stronger guarantees, likely at greater implementation cost. However, if the tools are too difficult to use, programmers may not obtain the guarantees because they may circumvent the tools (e.g., by implementing difficult procedures in a lower-level language) or because they may fail to complete their projects within their cost and time constraints.

In practice, there is typically a gap between what is actually specified in a formal specification of correctness and what is desired by the programmer. For example, a programmer may specify the correct output of a factorial function in a recursive way, implement the function iteratively to avoid overflowing the stack for large input, and leave unwritten the specification that *the program shall not overflow the stack for input within the expressible range of machine-size integers*.

2.3 Immutability Systems

Although there is a large collection of immutability-related systems proposed in the literature, we have not found any usability studies of these systems. A more comprehensive review of these can be found in prior work [156]. IGJ [223] implemented class- and object- immutability as well as read-only references and supported polymorphism, but did not enforce transitivity; this work included case studies but no user studies, so it is unknown whether other Java programmers can use IGJ effectively. Haack and Poll [88] proposed a type system for object immutability, read-only references, and class immutability that supported initialization outside constructors, but the only evaluation of this system was theoretical. Skoglund and Wrigstad [186] proposed a type system for read-only references in Java, but it only supported a subset of Java and it does not appear that they implemented their system. Java `final` and C/C++ `const` do not express transitive class immutability. .NET Freezable and JavaScript `Object.freeze` are enforced dynamically, not

statically.

Pure4j is an annotation processor for Java that provides `@ImmutableValue` [129], and, like Glacier (chapter 5), enforces that annotated classes are transitively immutable. However, it provides no solution for arrays, assuming that all arrays are mutable. It requires that all fields be declared `final`, which is redundant in Glacier. Unlike Glacier, it requires that public methods of immutable classes take only immutable parameters and that instance methods that are not inherited from a base class be pure. This is a stronger restriction than immutability and forbids access to global state, whereas immutability in Glacier pertains only to state reachable specifically via fields of objects. Although method purity can be helpful in certain cases, such as that of thread safety, it also restricts applicability. For example, a pure method cannot read or write files or the network. Our focus on immutability rather than purity reflects the evidence we have of what developers need (see chapter 4).

Immutables is another annotation processor for Java [120], but rather than enforcing immutability, it generates immutable classes from abstract value classes. It can also automatically generate builders and factories. Kjolstad et al. proposed a tool, Immutator [108], to automatically make immutable versions of classes and conducted an experiment showing that programmers make errors when refactoring classes to be immutable; our focus here was on enforcing immutability, not refactoring.

Some functional languages, such as Haskell, emphasize immutability. Isolating code that has side effects is a core part of Haskell, so it is unclear what the usability impact of this design decision is. Other functional languages, such as SML, promote immutable value types, but do not prevent mutable state or provide any way of forbidding it inside modules.

2.4 Blockchain Programming Languages

2.4.1 Smart contract languages

Solidity [73], which targets the Ethereum platform [72], is the dominant 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 [10, 59, 121], created static analyses for existing languages [103], and worked on applying formal verification [17]. Delmolino et al. [59] described a user study of Serpent, which was a precursor to Solidity, showing several classes of bugs that occurred in the lab. Among these was *asset loss*, which motivated our study to see whether our participants would avoid these bugs when using Obsidian (chapter 11). 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 [92]. One of the languages most closely related to Obsidian is Flint [178]. 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 [177]. 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. Another approach is to trade expressiveness for safety. For example, Pact [102] is Turing-incomplete, avoiding nonterminating behavior. Nomos [56] is a smart contract language based on session types that supports reasoning about resource consumption. This approach could be helpful on platforms (such as Solidity) that impose a cost semantics on computation, although the implementation does not support any particular blockchain platform.

Scilla [181] is an intermediate language for smart contracts. Like Obsidian, Scilla is oriented around state machines, but unlike Obsidian, it represents programs as communicating automata, avoiding complex inter-contract transactions (instead requiring that these be implemented as continuations). 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. Scilla's semantics were formalized in Coq. Likewise, IELE [106] 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 [173].

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 [98]. DAML [61] 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. [9] proposes a rules-based language to enable business users with less programming background to author smart contracts.

Xu et al. [219] 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.

2.5 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 [39]. Unfortunately, these approaches can be very complex. For example, fractional permissions [28] 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* [37]. Ownership types aim to enforce *encapsulation* by ensuring that the implementation of an object cannot leak outside its owner. Other approaches to ownership include *owners-as-accessors* [140], which views ownership as restricting access but not necessarily heap topology. In Obsidian, we are less concerned with

encapsulation or the heap 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. Because of this difference in meaning of *ownership*, we avoid using the term *ownership type* in relation to Obsidian.

Gordon et al. [80] 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 in order 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. [80] do not allow readonly (*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 [214], but have not been adopted in many programming languages. In *functional* languages, linearity may take the form of session types [32]. This approach mirrors typestate, since channel types (expressed as session types) change as messages are sent through them.

Although most programming languages do not support linear types, Rust [164] 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 [201] is an ML-like language that supports linear types; unlike Obsidian, it is not object-oriented. Session types [32] are another way of approaching linear types in programming languages, as in Concurrent C0 [216]. 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.

2.6 Typestate

Typestate was originally proposed by Strom and Yemini [194]. Drossopoulou et al. [65] introduced typestate for object-oriented languages in Fickle. In Fickle, objects could dynamically change class, but the static type did not reflect the changing classes. More recent work by DeLine, Aldrich, Bierhoff, and others describes how object-oriented languages can be used with typestate [5, 18, 58]. DeLine investigated using typestate in the context of object-oriented systems [58], finding that subclassing causes complicated issues of partial state changes; we avoid that problem by not supporting subclassing.

Plaid [198] and Plural [19] 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 [20]. Sunshine et al. [196] showed typestate to be helpful in documentation when users need to understand object protocols; we used that conclusion as motivation for our language design. Garcia et al. gave a formalization of typestate [78].

Chapter 3

PLIERS: A User-Centered Process for Programming Language Design[1]

3.1 Introduction

Programming languages serve as interfaces through which programmers and software engineers can create software. The ability of these users to achieve their goals, as with other kinds of interfaces, depends on the usability of the languages in which they do their work. For example, the presence of `null` in Java results in a particular kind of error-proneness, since programmers can easily accidentally write code that dereferences `null` [94]. These kinds of mistakes persist despite the training and experience that professional software engineers have.

The high-level research question discussed in this dissertation, *How can programming language designers leverage data from users to improve language usability?*, can be refined in terms of three research questions:

Naturalness: How can we obtain insights as to what language designs will be *natural* for programmers, given that we are trying to obtain particular static safety guarantees in the language?

Iteration: How can we iterate on a particular design so it continually gets to be more effective for users?

Comparison: How can we compare multiple language designs to see which are more effective for users?

Some authors, such as Stefik and Hanenberg, have focused on using quantitative approaches [188, 189]. Others have focused on in-depth case studies to evaluate their languages [3]. Our approach is to integrate a wide variety of both *qualitative* and *quantitative* user-focused methods with *formal theory-based methods* [152] to design programming languages [131]. This approach allows us to integrate user research into many different stages of the design process.

In order to address our three research questions, we adapted traditional HCI methods to the context of the design of programming languages that target professional software engineers. Then, we applied those adaptations to the design process of two new languages, Glacier and Obsidian,

[1] Portions of this chapter previously appeared in [45].

which we used as testbeds for language design methods. Finally, we synthesized the methodology into a process we call *PLIERS*: **P**rogramming **L**anguage **I**terative **E**valuation and **R**efinement **S**ystem. This chapter describes the methods and process we developed. Later, chapters 5, 8 and 11 will show we used PLIERS to develop and evaluate Glacier and Obsidian.

It is not enough to only use methods from HCI to design a programming language because, if a designer wants programs to have well-defined meanings (semantics) and well-understood safety properties (soundness), the programming language must also be subject to a collection of semantic constraints from the theory of programming languages. We have integrated strategic application of our adapted programming language design and evaluation methods into a process that also incorporates formal methods so that the resulting languages can be both usable and sound. Although the individual methods have been applied in the HCI literature in a variety of contexts, this chapter shows how we have adapted the methods and combined them to obtain insights regarding programming languages that target professional software engineers. PLIERS is not a recipe for language design, just as *agile* is not a recipe for software engineering. Instead, PLIERS provides a process for organizing language design work around human-centered methods. For each step in the design process, PLIERS suggests ways of integrating human-centered methods to inform the designers.

Designing programming languages requires expertise in type systems, compilers or interpreters, and language runtimes. As such, PLIERS is aimed at showing people with those technical tools that it is feasible and effective to include user-centered methods. By doing so, the goal is that the languages will be more effective for programmers than they might be otherwise.

In developing PLIERS, we focused on an engineering perspective on programming languages: namely, that programming languages are engineered artifacts whose purpose is to benefit users. There are other perspectives, however. Programming languages are means for exploring and manifesting mathematical ideas. They are also media for creation of software artifacts, and as such, they affect both the engineering properties of those artifacts *and* their aesthetics. Furthermore, even when there is scientific or mathematical evidence for certain design decisions, other decisions must be made on the basis of experience or even on the basis of beauty or aesthetics. PLIERS does not say how to make these kinds of decisions, how to make languages that are beautiful, or how to make languages in which programs will be beautiful. PLIERS can help lead to insights regarding users, but the concrete language design proposals remain a creative process. Thus, the role of creativity remains central to the language design process.

This chapter makes three main contributions:

1. We define the PLIERS programming language design process, which shows how user-centered methods can contribute to many different phases of programming language creation. We evaluated PLIERS by using it to develop two programming languages; we describe the benefits and areas for improvement that we observed in the process (§ 3.2.1).
2. We show how we have adapted several *formative* study techniques, such as natural programming, Wizard of Oz, rapid prototyping, cognitive dimensions of notations analysis, and interview studies to inform the design of Glacier and Obsidian. We found that our adapted methods were effective when used in the context of PLIERS.
3. We show how we conducted *summative* usability studies on new programming languages. By developing ways of teaching the languages efficiently, effectively, and consistently, we

were able to conduct usability studies of programmers using novel programming languages.

We designed PLIERS for use with language designs that require developers to learn challenging programming concepts or think in a new way. The safety properties that motivated the two languages we discuss in this dissertation provided opportunities for language constructs that could potentially be confusing, which made them ideal testbeds for PLIERS. However, other contexts provide other kinds of conceptual challenges for programmers, and PLIERS may be similarly useful in those contexts. For example, multicore programming results in concurrency challenges; distributed programming requires managing asynchronous requests that may fail; and domain-specific languages (DSLs) may require mastery of domain concepts. We expect that PLIERS will be useful for helping language designers with any language that requires programmers to master difficult concepts.

In developing PLIERS, we initially intended to apply known HCI methods, such as *natural programming* [132], *Wizard of Oz* [54], *interviews*, and *rapid prototyping*. However, we found that the study design process was very challenging due to the nature of programming and the complexity of the design space. For example, some of the challenges we faced included:

Recruiting: how could we recruit participants who have sufficient programming skill and whose results would generalize beyond the population of students, despite having limited access to professional software engineers?

Training: how could we train participants in a new programming language in a short enough amount of time to make studies practical?

High prototyping cost: how could we conduct user studies on programming languages that have only informal designs and no implementations, since the cost of building working prototypes is high?

Variance and external validity: how would we mitigate high variance, which is typical in programming tasks, without constraining the tasks so much that they were no longer representative of real-world programming tasks?

The problems of variance and external validity were particularly relevant for quantitative studies, which needed to be practical in the context of our university setting. Programming tasks that are *not* extremely constrained tend to produce results with high variance, making statistical significance hard to obtain. On the other hand, tasks that *are* highly constrained suffer from low external validity, since real-world programming tasks are typically long and complex.

Therefore, we had to modify existing methods to address these challenges. Our study design contributions are summarized in Table 3.1. For example:

- By adapting the *natural programming* technique to allow *progressive prompting*, we were able to obtain both unbiased responses as well as data that were relevant to the particular designs we were considering.
- By *back-porting* language design questions to languages with which participants were familiar and by using the *Wizard of Oz* evaluation technique, we were able to obtain usability insights on incomplete designs, and *isolate* the design questions of interest from confounding variables.
- By dividing large tasks into multiple, smaller tasks, and by using pilot studies to set task time limits effectively in quantitative studies, we were able to reduce variance sufficiently

Challenge	Approaches
Training	• Include knowledge assessments and practice problems in tutorial • Divide tutorial into small pieces • Answer questions during training phase of study • Automatically provide feedback for wrong answers
Recruiting	• In academic settings, recruit master's students, who frequently have professional experience that may be representative of many practitioners • Recruit professionals, but only when their expertise is needed • Appeal to professionals' altruism for recruiting (they may not be incentivized by typical study budgets) • Screen participants carefully; set a high bar for student participation • Evaluate language design research questions in the context of a language with which many possible participants are familiar
High prototyping cost	• Back-port language design questions to existing languages (also helps isolate effects of independent variables) • Use Wizard of Oz to simulate tools that do not exist yet: use a plain text editor rather than a real IDE, and have an experimenter provide feedback in lieu of a real compiler or interpreter
Interdependence of features	• Isolate design questions by back-porting them to a familiar language • Mitigate non-orthogonality risk with summative studies
Variance and external validity	• Triangulate with multiple study types • Break tasks into subtasks • Recruit from populations with sufficient programming skills and knowledge; pre-screen participants.
Time management	• Pilot repeatedly to assess how long tasks usually take • Set cutoff times so that most people will succeed at most tasks • Allow participants extra time when possible, then report these successes separately from the "within time limit" results
Bias toward familiar languages	• Staged natural programming approach: sequentially expose additional constraints to participants • Request that participants do tasks using specific language designs that are being evaluated
Unsound proposals	• Provide sound alternatives and ask participants to use them • Provide participants with expert feedback on design ideas

Table 3.1: How PLIERS addresses common challenges in running user studies on programming languages.

to obtain meaningful results in complex programming tasks.

- By recruiting participants who were *representative of at least some junior-level professional developers*, we were able to maximize external validity in our studies while still conducting them practically at a university setting. We were also able to show usability impacts of the language designs under consideration.
- By developing incremental tutorials with integrated practice opportunities, we were able to *teach* the languages to participants in a short time (for Obsidian, about 90 minutes was typical).

3.2 PLIERS

PLIERS is summarized in Figure 3.1. User-centered design methodology [87] seeks to leverage data from users to improve the design of systems. PLIERS is a specialization of user-centered design for programming languages to enable designers to incorporate ideas from user studies as well as from the theory of programming languages. PLIERS consists of five phases: need finding, design conception, risk analysis, design refinement, and assessment. In each phase, the designer seeks and leverages input from or about the users that the designer is hoping will use the programming language. If work in any phase calls into question the work done in a previous phase, the designer may return to the previous phase and conduct more work according to the difficulties that were identified.

Need finding: The process begins by assessing the user's needs. What programming problems does the user have, and how might a new programming language help the user achieve their programming goals? Some have advocated that language designers should design languages for their own use [82]. In contrast, PLIERS uses user-centered methods, such as a corpus study, interview, or contextual inquiry to understand the target audience and what their needs are. The designer chooses which particular user-centered methods to use according to the available resources and the goals of the design project. These user needs may be stated as hypotheses regarding what kinds of languages might benefit users, what *benefit* means to those users, and how those benefits might be assessed.

Design conception: After assessing users' needs, the designer must conceive of initial language ideas that might satisfy those needs. As in other design situations, this process often requires significant creativity. The designer iterates between two kinds of work: theoretical work, in which the designer develops a theoretical foundation for the programming language (a *core calculus*), and prototyping work, in which the programmer directly works on the language that programmers will see (the *surface language*). The result of this work is expressed as a low-fidelity prototype, such as a corpus of code samples (to demonstrate the surface syntax, by which users will write and edit programs) and a core calculus.

Risk analysis: In the risk analysis phase, the designer assesses and prioritizes usability risks in the proposed design. This step leverages theoretical models of cognition and usability inspection techniques to identify the aspects of the language design that are most likely to present difficulties to users. For example, cognitive dimensions of notations [85] can help identify usability risks that are worthy of further study.

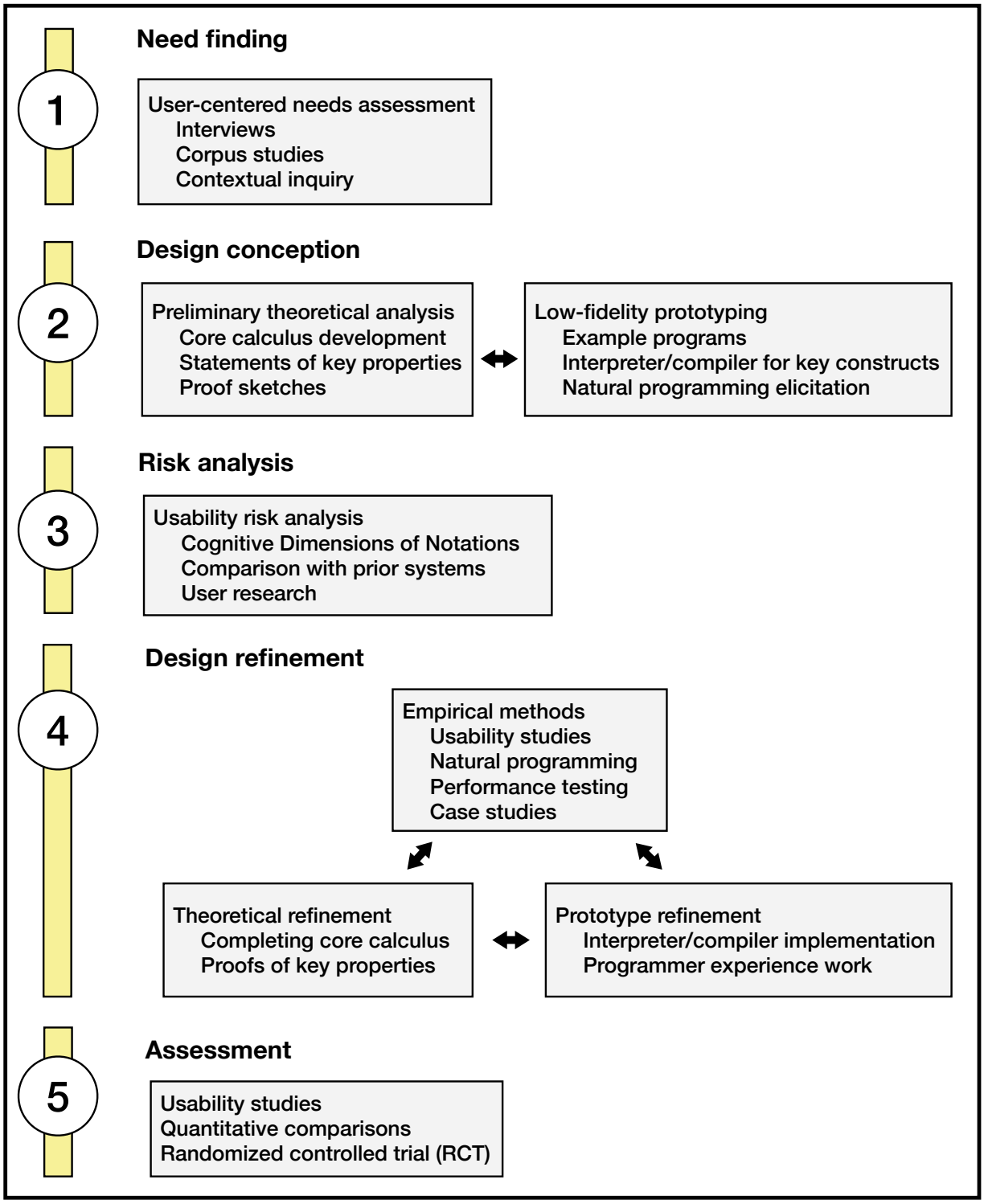

Figure 3.1: The phases of the PLIERS process, showing activities conducted in each phase. Designers can return to previous phases if evaluation identifies opportunities for improvement.

User research might be needed in this phase to better understand the target audience. For example, if the designer is considering an approach that requires particular skills, then risk analysis might include assessing to what extent the target audience has those skills or whether those skills can be taught in an acceptable amount of time. If the language targets professionals, substantial training may be acceptable. If the language targets end-user programmers, the designer may want to limit the training that would be required.

The prototype design likely leverages some elements of existing languages, while creating new features that may be unfamiliar to users and have unknown usability characteristics. These new features may be particularly worth evaluating. Each risk corresponds to a design or research question, and provides an opportunity for learning more about how to make the programming language as usable as possible.

Design refinement: Using empirical methods, the designer assesses the usability risks identified in the prior phase. Then, the designer refines the prototype, successively increasing prototype fidelity as usability risks are addressed. The designer also considers other language requirements, such as *expressiveness* (can the language be used to write a particular kind of program?) and *performance* (does the program, when run, meet the designer's performance goals?). Then, the results are used to revise the theoretical model and the prototype. By using theory, the designer can ensure that any changes retain any formal guarantees that the language promises. Alternatively, the designer may choose to guarantee different properties in order to allow the desired modifications.

Because the theoretical model and prototype are related, changes in one frequently lead to changes in the other. Eventually, the theoretical analysis will include proofs of key properties, and the prototype will be high-fidelity, typically including IDE support and a compiler or interpreter.

Assessment: A summative usability study can assess whether the final design has achieved the designers' usability objectives. In contrast to the usability studies in the previous phase, which assess specific aspects of the language design, a summative study is intended to assess programmers' abilities to complete realistic programming tasks.

In this thesis, we focus on usability-related objectives, but the designer may want to conduct performance testing as well. For performance evaluation, we refer readers to the SIGPLAN empirical evaluation checklist [13].

In developing the PLIERS process, we identified a collection of adaptations to traditional HCI methods, which helped us obtain useful information, primarily in the *design refinement* and *assessment* phases. Here, we describe some of the key ways in which we modified existing methods, which are further discussed in chapter 12 as well as other chapters. The methods we have found useful are summarized in Table 3.2.

Back-porting design questions to existing languages: To study the usability of a design decision in isolation, we start from an existing language with which participants would already be familiar. For example, rather than asking participants in our early formative studies to learn a whole programming language, we told them that we were adding certain features to Java, and then asked them to complete programming tasks in the Java variant. This substantially reduced the training time and allowed us to reason that any confusion was

Design Task	Method or Component	Details
Need finding and hypothesis generation	Interviews	• Interview practitioners (e.g., software engineers) to identify problems and formulate hypotheses
	Corpus analysis	• Analyze corpora of applications and bugs to identify common goals and obstacles
Formative design evaluation	Natural programming elicitation	• Ask participants to do programming problems without giving them syntax or identifiers in order to help design a syntax and vocabulary that matches their expectations
	Programming tasks	• Back-port design components to an existing language to isolate variables of interest • Break larger tasks into subtasks to constrain unproductive exploration • Include a range of task difficulties to obtain data from both more- and less- successful participants • Use low-fidelity prototypes to obtain early feedback on designs • Use Wizard of Oz to enable studies of incomplete prototypes
Summative design evaluation	Usability studies	• Assess what barriers users face when attempting to complete relevant programming tasks
	Randomized controlled trials (RCTs)	• Compare task times and success rates between different languages

Table 3.2: A summary of user-centered methods that we have found useful for studies in PLIERS.

likely related to the new features, since our participants were already familiar with Java.

Selecting a target language for back-porting depends on several factors:

1. Availability of participants skilled in the target language
2. High-level similarity between the novel language and the target language (e.g., both object-oriented, both functional, etc.)
3. For high-fidelity prototypes: development cost of modifications to target language

We also favored high-level design decisions that allowed us to attract participants who had relevant background. If we had tried to teach participants a completely novel language paradigm even though the basic assumptions of the language paradigm were not the targets of our research, we would have needed to try to distinguish the *relevant* mistakes from all the novice-level mistakes that the new programmers would be likely to make.

Wizard of Oz: Implementing a programming language is expensive. Rather than implementing a full compiler for each language variant we wanted to test, we adapted the Wizard of Oz technique [54]. In a classic Wizard of Oz study, an experimenter pretends a system is working by remote-controlling it in order to obtain insights about potential designs without having to actually build the system.

In early Obsidian studies, we gave participants a text editor, documentation, and programming tasks to do. Then, an experimenter verbally simulated compiler errors. Like a modern IDE, the experimenter could interject with errors, and could provide error messages when participants asked whether the compiler would emit any errors on their current code. For example, the experimenter might say "Suppose your compiler indicated that there was asset loss that occurred on line 42." If the error messages were unclear, the experimenter could revise them with more detail, helping us understand how to write clear error messages for the compiler. The technique we developed allowed efficient iteration on our design ideas, since design changes only required updating the documentation, not a potentially complex implementation. Unlike in a traditional Wizard of Oz study, participants were aware that the feedback was being provided manually, but we observed that this did not present an obstacle to the effectiveness of the technique.

Multi-part tutorials: In our studies, we needed to teach participants a new programming language in a consistent and efficient way. A traditional course would not be effective, since we could not recruit our participants into a semester-long course. Instead, we developed multi-part language tutorials. By breaking the tutorial into sections (each about ten minutes long), including practice problems for participants to do, and having an experimenter available to answer questions, we were able to convey the knowledge we needed in a relatively short period of time. Our longest tutorial, for example, took participants an average of 1 hour, 35 minutes.

3.2.1 Evaluating PLIERS

To evaluate PLIERS, one might like to teach PLIERS to a collection of programming language designers and conduct a qualitative study regarding the insights the designers obtained. Better yet, one might like to recruit language designers and assign them to either use or not use PLIERS

to design a language in a domain, and then conduct usability studies of the resulting languages. Unfortunately, these approaches are impractical: language design and implementation is typically a long process, taking months or years, and language designers cannot be recruited for such studies.

Another approach might be to teach PLIERS to designers who have recently completed their designs and observe what changes the designers make as a result. However, a main benefit of PLIERS is that it provides a framework for the entire language creation process; many of the methods can be applied to incomplete prototypes or design concepts. Such an analysis, though useful, would only obtain insights on some of the components of PLIERS.

Because of these practical considerations, we evaluated PLIERS by using it ourselves to create Glacier and Obsidian. In the process, we observed how the approach helped us create and iterate on the language designs. This approach has significant limitations. Our evaluation does not show that other designers can use the process effectively, that it works on a wide variety of different programming languages, that languages produced with the method are necessarily superior to languages produced without the method, or even that the process does not make languages *worse*. However, using PLIERS ourselves was a necessary part of developing the process; in this dissertation, we leverage our experience creating PLIERS in the hope that others may benefit from it and iterate on its component methods. As this is the first work of which we are aware that integrates HCI methods into the design of languages for professional software engineers, we view this is the first step toward creating a design process that will aid designers in making professional-level languages more usable.

Venable et al. provide a framework for evaluating design methodologies [209]. In that framework, our approach to evaluating PLIERS would be considered *ex ante* (formative, evaluating the design method before it is complete). The ex ante approach was appropriate for PLIERS since the work was conducted in large part to create and iterate on PLIERS. The evaluation included some aspects of *naturalistic* evaluations (it was evaluated in the context of real language design projects) and some aspects of *artificial* evaluations (the designers of the method used the method instead of third parties). This choice was driven by the impracticality of recruiting programming language designers other than ourselves to use our process over the long period of time required to conduct a language design and implementation project.

Blandford and Green have acknowledged the lack of an established path to acceptance for new methods and the difficulty of conducting rigorous evaluations of design methods [22]. We regard our work as the first step of many in evaluating PLIERS.

Chapter 4

Exploring Immutability Features[1]

4.1 Introduction

Many designers of APIs and programming languages recommend using immutability in order to prevent bugs and security flaws. For example, Bloch devoted a section of his book *Effective Java* [23] to minimizing mutability. He cited the following benefits of immutability: simple state management; thread-safety; and safe and efficient sharing. Oracle's *Secure Coding Guidelines for Java SE* [165] states that immutability aids security. Microsoft's *Framework Design Guidelines* recommends against defining mutable value types in part because they are passed by copy, and programmers might write code that modifies copies but should instead modify the original structure [163]. Some programming languages, such as Rust [164], take these recommendations into account by carefully managing mutability. The functional programming language community is particularly concerned with state management, producing languages such as Haskell, which segregates code that manipulates state from code that does not. Proponents of functional languages argue that avoiding mutable data structures facilitates reasoning about behavior of programs because one can reason equationally about behavior rather than needing to know about the global or even local program state [1].

There are questions, however, about what immutability should mean and how to express immutability in programming languages, as evidenced by the myriad of different kinds of support that tools provide and the lack of empirical data to justify specific design choices. Some languages support programmer-provided specifications of immutability (expressing that certain data structures cannot be changed) or read-only restriction (expressing that certain references cannot be used to change a data structure). For example, `final` in Java can express that a particular field cannot be reassigned to refer to a different object, but the contents of the referenced object may still change, and there may be non-`final` references to the same object. Furthermore, there is no way to express class-level immutability directly.

In C++, `const` data can still refer to non-`const` data, which can then be changed. When used on pointers, similar to `final`, `const` provides no guarantees regarding *other* references to the same object. This means that in addition to not providing the expected benefits of immutability to programmers, such as thread safety and simple state management, these annotations also do

[1]Portions of this chapter previously appeared as [49, 50].

not provide the guarantees that would be needed for the many compiler optimizations that require all of an object's behavior to be guaranteed to be fixed.

This chapter makes the following contributions:

1. After reviewing the existing literature and implemented systems in this area, we develop a classification system for mutability restrictions in programming languages (§ 4.2), show where some existing systems fit in this classification (§ 4.3), and discuss possible usability implications of existing design choices (§ 4.4).
2. We interviewed eight expert software engineers to find out how practitioners feel about existing language features and tools, and what their perceived needs are. We show that existing programming language features and research tools do not adequately address the issues encountered by practitioners. We extract design recommendations from the results of our interviews (§ 4.5).
3. IGJ is a Java extension that adds annotations specifying immutability and read-only restrictions [223]. We describe the design of *IGJ-T*, an extension to IGJ that enforces *transitive immutability*, which addresses problems that our interview participants described. We iteratively evaluated IGJ-T with three pilot users and refined our study based on the feedback (§ 4.6). The next chapter, chapter 5, describes Glacier, which addresses the problems we identified in the user study.

4.2 Overview of Concepts

It is important to distinguish among the many different meanings given to the term *immutability* and related concepts. In this section, we will give an overview of the various mechanisms and issues. Up to this point, we have been using the term *immutability* informally; here we synthesize definitions from existing literature to form a definition that will be used in the rest of the dissertation. A summary of the concepts appears in Table 4.1.

We use *object* here to mean any kind of state, such as a C `struct` or a functional language ref cell. *State* is data that is stored in memory. As an abbreviation, we will use *function* to refer to both functions and methods.

4.2.1 Type of Restriction

Immutability restricts modification through *all* references to a given object (see § 4.2.5 for exceptions). In contrast, *read-only* restrictions disallow modification through *read-only* references to a given object, but not necessarily through all references. The distinction between read-only restrictions and immutability is critical to the correctness of programs. For example, an immutable object can be shared safely among threads without locks, but a function that has a read-only reference to an object has no guarantee that the object cannot be modified, so that function must assume the object may be mutable.

Assignability restrictions, such as `final` in Java, disallow assignment. In most imperative languages, variables are backed by storage, so assignability restrictions represent non-transitive immutability (see § 4.2.3). Java's `final` keyword on fields is an *assignability* restriction.

Although a `final` field can never point to a different object than the one it was initialized to point to, the referenced object's fields may themselves still be modified. In contrast, the C declaration `const int *x` provides *read-only* restrictions: `x` is a pointer to `int` and might later refer to some other address, but the reference `x` cannot be used to change the value at any memory location to which `x` points.

Ownership systems define a system-specific notion of ownership and a way of specifying which objects a given object owns. This enables enforcement of restrictions such as "an object may only directly modify objects it owns". *Ownership types* [38] use a notion of object context to partition the object store. This partitioning allows the systems to ensure that aliases to objects do not escape their owners, ensuring representation containment and defining a notion of abstract state, since the abstract state (§ 4.2.5) of an object includes only data it owns. Ownership is also useful for ensuring thread safety of mutable structures by ensuring that locks are acquired in a correct order according to the ownership structure before accessing objects [27].

4.2.2 Scope

Object-based restrictions apply to a particular object. *Class-based* restrictions apply to all of a class's instances. For example, `final` is object-based: it only applies to a specific reference and there is no way of specifying that all references to instances of a particular class are `final`.

Class restrictions, while commonly either only supported as syntactic sugar for object restrictions (e.g., IGJ) or unsupported entirely, are frequently needed in practice according to our interview participants. By necessity, many programmers who want class immutability must improvise (§ 4.3.2).

4.2.3 Transitivity

Transitive restrictions pertain to all objects reachable from a given object, including objects captured by closures or methods. *Non-transitive* restrictions pertain only to the immediate fields in a given object.

Non-transitive restrictions provide weak guarantees because they say little about the *behavior* of the abstraction that the object's interface provides. For example, if a list object is non-transitively immutable, then the number of items in the list is fixed, and the list always refers to the same objects, but the contents of those objects can still be changed. A function that needed to know that a list of integers only contained positive integers would need to re-check all elements after every possible opportunity for mutation of list elements. However, in order to ensure that an object is immutable, one must verify that all objects in the transitive closure of references from that object are immutable.

Though the assignability and read-only features provided in many popularly-used languages are non-transitive, including Java's `final` and C++'s `const`, researchers have proposed transitive restrictions in a variety of different systems. Because of the important implications of this design decision and the discrepancy between research and practice, we will focus on this question in § 4.6.1.

4.2.4 Initialization

Systems might *relax* restrictions during initialization in order to facilitate initialization. A common method for creating a cyclic data structure involves modifying an element after it is created. The cyclic data structure may be mutable during initialization but immutable afterward. Alternatively, systems can *enforce* restrictions during initialization.

4.2.5 Abstract vs. Concrete State

The *abstract state* of an object refers to the portion of the state of an object that is conceptually a part of that object. For example, in a splay tree, sometimes during a read operation the internal tree structure will be rotated in order to move frequently accessed elements closer to the root. Even though this is a change to the internal data structure, it is not exposed to the caller and therefore read operations do not change the splay tree's abstract state. In contrast, the *concrete state* includes the object's entire representation.

Caches are often excluded from the abstract state because the contents of the cache duplicate information available elsewhere, and other than causing performance differences, the contents of the cache are invisible to clients. The abstract state might also exclude state kept only for debugging purposes. By excluding the cache from the abstract state of the object, writing to the cache can be considered a *benign* operation and can therefore be done by clients without write access. However, this allows the possibility of race conditions if the cache is not thread-safe but a programmer assumes it is because the object appears to be immutable.

Logical restrictions pertain to an object's abstract state. *Bitwise* restrictions govern an object's concrete state; this term arose in C, where programmers consider how structures are arranged in memory. Some languages provide features that let programmers differentiate these: in C++, for example, a `mutable` member variable can be modified even through a `const` reference. Unfortunately, it may be tempting for a programmer to assume that if all references to an object are `const`, the object is thread-safe, but the `mutable` members may be mutated in a thread-unsafe way. Thus, an object that is only logically-immutable may not be thread-safe.

4.2.6 Backward Compatibility

When facilities for restrictions are added to a programming language after the language is created, code that uses the extended features may need to interface with code that does not. If this is to be permitted, then there is a question of what guarantees are made. For example, if a reference to an immutable object is passed as input to an function whose interface does not specify restrictions, then the called function might mutate the object, violating the guarantee. Systems that make a *closed world* assumption assume that all code that uses code with restrictions itself has any restrictions declared. In contrast, the *open world* assumption is that there may be interfaces with un-restricted code and that the system must provide guarantees for this code too, either by making conservative assumptions about these APIs or by checking conditions dynamically. Open-world systems that support class immutability must ensure that instances of immutable objects encapsulate their representations because otherwise clients may mutate the representation

objects directly. Furthermore, open-world systems that support object immutability must ensure that immutable objects are not exposed to unchecked clients [89].

The *closed-world* assumption can be a significant impediment to adoption for language extensions, since in a system that makes a closed-world assumption, adoption in new code requires that all clients of that code also adopt the system.

4.2.7 Enforcement

Static restrictions are enforced at compile time. *Dynamic* restrictions are enforced at runtime.

Static enforcement typically has a problem of *virality*: in order to call a method on an object that has an immutability or read-only restriction, the method typically must guarantee that it will not modify state, but if the method itself calls methods, then those methods must also be so guaranteed, and so on for the transitive closure of methods called by the first method. This can be burdensome if the guarantees must be annotated by programmers. In addition, the static analysis must be conservative, and therefore may give errors on code that is actually safe. C++'s `const` is viral, and programmers complain that "const-correctness" is therefore difficult to achieve.

Anders Hejlsberg, the lead C# architect, when asked to explain why C# did not offer C++'s `const` feature, stated the problem quite bluntly:

> We hear that complaint all the time too: "Why don't you have const?" Implicit in the question is, "Why don't you have const that is enforced by the runtime?" ...
>
> The reason that `const` works in C++ is because you can cast it away ... [210]

4.2.8 Polymorphism

Restriction polymorphism would mean that the same function can operate on inputs with different restrictions. Parametric polymorphism in restrictions is the most relevant example: restrictions can be specified in a parameter rather than explicitly so that one implementation can operate on inputs with different restrictions, while still obeying those restrictions. *Non-polymorphic* restrictions require programmers to write implementations that statically assume particular restrictions.

In C++, `const` is monomorphic: each method's parameters and result are either `const` or not. The result is a common pattern: the programmer must write `const`- and non-`const` versions of many methods so that those methods can be invoked with both `const` and non-`const` inputs. For example, in C++, it is impossible to write a single "identity" method that takes a `const` or non-`const` array and returns the same array with the access restriction preserved; one must instead write two (typically overloaded) different methods. In contrast, IGJ [223] supports immutability parameters `@I` so that the restrictions can be expressed as a parameter of a type.

4.3 A Survey of Existing Systems

Table 4.2 summarizes some related systems. The table has a focus on object-oriented and imperative systems; since mutation tends to be ubiquitous in these kinds of systems, these seemed to offer the greatest opportunity for improvement by restricting mutation. In contrast, functional languages tend to result in less common use of mutable state.

Dimension	Possible choices
Type	**i**mmutability, **r**ead-only restriction, **a**ssignability, **o**wnership
Scope	**o**bject-based, **c**lass-based
Transitivity	**t**ransitive, **n**on-transitive
Initialization	**r**elaxed, **e**nforced
Abstraction	**a**bstract, **c**oncrete
Backward compat.	**o**pen-world, **c**losed-world
Enforcement	**s**tatic, **d**ynamic
Polymorphism	**p**olymorphic, **n**on-polymorphic

Table 4.1: Summary of Dimensions

System	Type	Scope	Trans.	Init.	Abstr.	Compat.	Enforc.	Poly.
Java `final`	a	o	n	e	N/A	c	s	n
C++ `const`	r	o	n	e	a	c	s[1]	n
Obj-C immutable collections	i	o	n	e	N/A	c	s[1]	n
.NET Freezable [126]	i	o	n	e	N/A	o	d	n
Java `unmodifiableList`	r	o	n	e	N/A	o	d	n
Guava `ImmutableCollection`	i	o	n	e	N/A	o	s, d[2]	n
IGJ [223]	i, r	c, o	n	e	a	c	s	p
JAC [109]	r	o	t	e	c	c	s	n
Javari [202]	r	c, o	t	e	a	c	s	p
OIGJ [224]	i, r, o	c, o	n	r	a	c	s	p
`immutable` [89]	i, r, o	c, o	t	r	a	o	s	p
C# isolation extension [80]	i, r, o	c, o	t	r	a	c	s	p
JavaScript `Object.freeze`	i	o	n	e	c	o	d	n

Table 4.2: Summary of Some Existing Systems (abbreviations are from Table 4.1)

[1] These approaches provide static enforcement to the extent possible in these languages.

[2] Static deprecation warning, runtime exception

4.3.1 Historical and Research Systems

The functional programming community was one of the first proponents of *immutability*. In 1984 Abelson and Sussman [1] promoted immutability on the grounds that it supports formal reasoning and makes concurrent programming easier. Programs that use only immutable structures are called *pure*. Functional languages have a variety of approaches to restricting mutability. For example, Haskell uses monads to express changes in state rather than allowing any kind of direct ref cell manipulation (as is permitted by SML). By doing so, operations that have effects must specify those effects in their signatures. Another way to achieve the goal of exposing effects is with type and effect systems [139], which extend type systems with information regarding side effects, such as potential writes to regions of memory.

In contrast to functional languages, imperative languages emphasize mutation of program state. By adding features that allowed restrictions on what data structures could be modified in each context, language designers hoped to facilitate reasoning about programs. At first, languages provided features restricting what functions could do to their parameters. In Pascal [8], `var` parameters were passed in by reference, but non-`var` parameters were passed by value, preventing modification of the passed parameter. Modula-3 [34] provided a `READONLY` parameter annotation, which prevented a function from using the formal parameter in a context that would require an l-value, such as the left-side of an assignment statement. In Ada [7] parameters to functions could be declared `in`, `out` or `in out` to indicate whether the function could read but not modify, modify but not read, or both read and modify each parameter.

Later, support for modularity led to features that controlled mutation in entire modules. In Turing [96], a variable defined in module A could be imported non-`var` into module B which would prevent the variable from being changed by B, although A could still mutate it. A common attribute of these features was that data structure mutation would be permitted through some references but not others, i.e., these represented read-only restrictions.

IGJ [223] provides Java annotations that implement immutability. For example, `@Immutable Date d` is a reference to an immutable date. No fields can be modified on an `@Immutable` object; IGJ verifies that no non-`@Immutable` references to an `@Immutable` object can be obtained. `@Immutable` specifies non-transitive immutability: if an `@Immutable` object's fields are not `@Immutable`, then those objects' fields may still be assigned to. `@Mutable`, the default annotation for unannotated fields, also grants exceptions permitting modification of fields in `@Immutable` objects.

`@Readonly` in IGJ specifies a read-only reference. The holder of a `@Readonly` reference cannot use that reference to modify the referenced object, but there may be other non-`@Readonly` references to the same object (*aliases*). IGJ also supports a form of class immutability, in which specifying `@Immutable` on a class serves as syntactic sugar in place of adding annotations in a variety of other places. Finally, IGJ supports an immutability parameter `@I`, which takes the value of another annotation. For example, if `@I` is the immutability parameter for a class and a field in that class is annotated with `@I`, then that field in a particular instance of the class will have an immutablity annotation according to the annotation of that instance. This is the sense in which IGJ supports transitive immutability: if all fields are annotated with the class's immutability parameter for all fields transitively included in that class, then immutability specified by a reference to an object will be transitive.

Unkel and Lam [206] generalized `final` to define *stationary* fields, for which all writes occur before all reads, and provided an algorithm to find them. Of fields in their corpus, 44-59% were stationary, but only 11-17% were `final`.

Haack et al. designed a Java-like language with a class modifier, `immutable`, which specifies that all class instances are immutable, and uses ownership types (with ownership annotations) to enforce encapsulation [89]. Later work by Haack and Poll [88] avoided the need for ownership types and supported object immutability, read-only references, and class immutability. They also permitted temporary modification of read-only structures after initialization by using stack-local memory regions to isolate new immutable objects.

Skoglund and Wrigstad [186] proposed a transitive read-only restriction system for Java, which includes the ability to check for read-only restrictions at runtime. Zibin et al. [223] argued that allowing runtime checks of restrictions hampers program understanding but there are no user studies confirming or refuting this claim. JAC [109] provides `readonly` (a transitive read-only restriction) as well as `readimmutable` (a transitive read-only restriction that also disallows reading of transitively mutable state) and `readnothing` (providing no access to state at all; suitable for pure functions). JAC extracts a readonly portion of each class by restricting return types: methods return read-only references when called on read-only objects. As a result, a programmer need only provide one implementation of a class, and JAC can generate restricted versions of the class. However, this approach has not been adopted by the Java community. In contrast, Javari [202] provides a simpler access rights system (for example, JAC includes three different levels of read-only access, but Javari only has one).

One impediment to understanding and reasoning about programs is the general alias problem of determining what references exist to a given object. With a precise alias analysis, one could find out whether a given object might be modified by a particular call. Unfortunately, a precise *may-alias* analysis is undecidable [115]. Therefore, various approaches have been developed that facilitate reasoning about aliases by restricting which aliases can exist.

Noble, Vitek, and Potter [141] presented a system for *flexible alias protection* that provides several aliasing mode declarations that allow aliasing invariants to be checked statically. For example, one mode ensures that "a container's representation objects may be read and written, but must not be exposed outside their enclosing container…". Other approaches for alias protection include balloon types [6] and islands [95], which prevent external references to objects that are encapsulated by other objects' interfaces. Servetto et al. proposed *placeholders* as a technique for safely initializing circular immutable structures in an extension to Featherweight Java [182].

Ownership types, introduced by Clarke [38], use a notion of object context to partition the object store. Many of these, including Clarke's original approach, restrict aliasing. Other approaches, such as universe types [60], restrict mutation but not access to owned objects. In contrast, Rust [164] expresses ownership without explicit object contexts; instead, it works at the level of variable bindings to ensure that no more than one binding exists to a given resource. Rust provides facilities for borrowing ownership at function call boundaries in order to make function calls more convenient. The ownership system OIGJ [224] extends IGJ with a notion of ownership so that objects cannot leak outside their owners. Potanin's chapter on immutability [156] gives more detail on ownership types and immutability in general.

Gordon et al. [80] focused on providing safe parallelism by combining immutable and isolated types, with support for polymorphism over type qualifiers. They provided `writable`,

`readable`, `immutable`, and `isolated` qualifiers. An `isolated` reference ensures uniqueness: "all paths to non-immutable objects reachable from the isolated reference pass through the isolated reference."

Another approach is to use *capabilities*, which pair pointers with policy information that specifies what access rights accompany those pointers [29]. This changes the default on references from "no restrictions" to "all restrictions" but specifies all attributes positively, for example permitting writing. This approach is in contrast to all of the approaches described above, which use language features to express restrictions relative to a default of no restrictions.

4.3.2 Popular Languages and Libraries

We show through examples below that despite the fact that some popular languages offer read-only restrictions, programmers still desire immutability guarantees and attempt to implement them using the features that they have available. This often leads to misunderstandings, if not by the original programmer, then by other programmers that have to read and maintain the code.

For example, the CERT C Secure Coding Standard [179] says "Immutable objects should be `const`-qualified. Enforcing object immutability using `const` qualification helps ensure the correctness and security of applications." C does not have any features that guarantee immutability in general, though `const` does provide some guarantees for fields. In particular, if one has a pointer allowing `const` access to an object, there may be other non-`const` pointers to that same object. For example, consider this C code, which uses `const` to express constraints in an interface:

```
void threadUnsafePrintIfPositive(const int *x) {
  if (*x > 0) {
    printf("%d", *x);
  }
}
```

Even though `x` is a pointer to a `const int`, this only means that `threadUnsafePrintIfPositive` cannot modify the referenced `int`. This function is not thread safe because it dereferences `x` twice and `*x` may change between the dereferences. If `*x` were immutable rather than just read-only, then the above code would be safe, but such immutability cannot be expressed in C.

As we described in § 4.2.1, the requirements for making a Java class immutable are complex. If a method returns a reference to an internal data structure and cannot ensure that it cannot be modified by a caller, the structure must be copied before being returned to prevent a caller from being able to mutate it. This is often easy for programmers to ignore or forget, and the results in security-sensitive code can be serious. For example, Java's "Magic Coat" security bug was caused by the `getSigners()` method returning a reference to a mutable array holding the signers of a class [124].

Rather than supporting immutability or read-only restrictions in programming languages, some authors have added limited support in libraries. These typically convey these restrictions in names and documentation, resulting in ad hoc (i.e., expressed informally, rather than formally in language constructs) class-based restrictions. The result is that a client for whom immutability of

a data structure is required cannot rely on the compiler to give an error if the data structure is later made mutable.

For example, the Foundation framework [159] provided with Objective-C separates immutable and mutable classes in many cases by making the mutable classes subclasses of the immutable classes: the subclasses have additional methods that expose mutation. This ensures that mutating methods cannot be invoked on immutable objects (except that Objective-C is not type-safe). Likewise, the Java JDK provides a collection of immutable classes, such as String and Number, but these are not specified as being immutable in any formal way. In Java, to make a class immutable, Bloch [23] recommends doing all of the following: provide no methods that modify an object's state; prevent subclassing; make all fields final and private; and ensure exclusive access to mutable components. If a programmer wants to know whether a class is immutable, complicated manual (but mechanizable) verification is necessary, but even this does not guarantee that the class will be immutable in the future, since a *new version of the code can easily make the class mutable* in a way that does not cause the compiler to produce any error messages for clients.

The Java JDK provides `Collections.unmodifiableList`, which wraps a given collection object with a wrapper that throws exceptions on modification. However, modifications to the original list are still permitted, so this approach is an example of a read-only restriction, not immutability. Furthermore, the objects in the list may still be modified, so the restriction is non-transitive. In contrast, Google's Guava library copies all the elements when constructing an `ImmutableCollection` from a `Collection`. However, `ImmutableCollection` retains the mutating methods from its superclass, so though the compiler gives a deprecation warning, code that attempts to modify an `ImmutableCollection` compiles and raises exceptions at runtime. Furthermore, `ImmutableCollection` can contain mutable objects, so the immutability is non-transitive.

Microsoft's .NET framework provides a Freezable class [126], which is a superclass for objects that can have a state in which they are immutable. The author of a class that descends from Freezable must add calls to specific APIs before and after modifying state, so enforcement is dynamic and semantics depend on the placement of those calls. Similarly, JavaScript includes `Object.freeze`, which dynamically enforces shallow immutability [130].

4.3.3 Empirical Evaluations

Though empirical studies of the effects of mutability on programmer productivity and program comprehension have been conducted, they have been quite limited. For example, Dolado et al. [63] provided initial empirical data regarding the influence of side effects, which mutate data structures, on program comprehension. They compared pre- and post-increment to explicit assignment in C, finding that programmers answered questions about program behavior more accurately with explicit assignment than with pre- and post-increment (e.g., `x=y+1; y=y+1` vs. `x=++y`). However, this comparison was limited to properties of syntax and does not provide recommendations for other design questions, such as on how to enforce encapsulation.

Stylos and Clarke [195] examined the process by which programmers write code that instantiates objects in C++, C#, and VB.NET and found that users prefer and are more effective at instantiating *mutable* classes. When instantiating immutable classes, the programmer must supply all parameters to the constructor at initialization time (the *required constructor* pattern);

in contrast, with mutable classes, it is possible to supply none or only some of the parameters at initialization time and defer setting the rest until later (the *create-set-call* pattern). When figuring out how to call the constructor of an immutable class, programmers must interrupt their work to figure out how to instantiate the arguments, which may themselves have arguments, and so on. The create-set-call pattern, in contrast to the required constructor pattern, lets participants defer understanding how to create the required arguments until after they had finished calling the first constructor. The authors concluded that in contrast with the common advice to prefer immutability over mutability, immutability sometimes interferes with usability.

The paucity of empirical work in this area motivated us to conduct our own interviews with programmers. Our aim was to find out how immutability-related language constructs affect programmers (see § 4.5 below). However, significantly more work will be required in this area if we are to base language design decisions on empirical data.

4.4 Usability of Features

The large collection of different possible language features supporting state change restrictions presents an interesting design problem. Many of the features are compatible with each other and, indeed, more recent systems have included a collection of different features so that programmers can choose which features to use. However, including all of the possible features at the same time makes a system more complex, and more complex systems are harder to use. For example, one of our interview subjects mentioned that in C++, it is common practice to only use `const` for pointers to `const` objects and never for `const` pointers to mutable objects because it's too hard to keep the distinction straight. `const int * x`, `int const * x`, `int * const x`, and `int const * const x` are all legal C declarations; the first two denote pointers to const integers, the third denotes a const pointer to a variable integer, and the fourth denotes a constant pointer to a const integer.

It might seem that a more expressive language — one that allows the programmer to be more precise about what invariants should be checked — would always be better than a less-expressive language. However, this is not necessarily the case. The Cognitive Dimensions of Notations framework [85] provides guidelines for evaluating and comparing usability of different notations. The *error-proneness* dimension refers to the probability of making mistakes, particularly ones where the consequences are hard to find. Language features for restricting state change exist primarily to *prevent* bugs and *clarify* meanings. However, in many cases, using the wrong restriction results in a weaker guarantee than intended but no obvious immediate problems. For example, in a situation that requires object immutability, the programmer might specify a read-only restriction instead by mistake. This may typecheck but not provide the needed guarantee. Likewise, a programmer might annotate an interface as returning a read-only object when in fact the returned object is immutable. This might lead clients of the interface to go to extra trouble and degrade performance, such as by adding locks to ensure thread-safety, when the object was already immutable.

In contrast to the disadvantages of complexity, the *hidden dependencies* dimension of Cognitive Dimensions, which refers to problems that occur when dependencies are not obvious, confirms a positive aspect of state change restrictions: when a reference to mutable state exists, a programmer

may write code that mutates that state without being aware of the reference. This situation reflects a hidden dependency, so this dimension suggests not only that immutability is likely to be helpful, but also that transitive immutability is likely to be better than non-transitive immutability.

Nielsen's heuristic evaluation technique [138] offers a collection of heuristics that can be applied when evaluating user interface designs. These heuristics are particularly useful when user studies with real users cannot be conducted, perhaps due to an incomplete implementation or cost constraints. The "be consistent" heuristic, which is also included in the Cognitive Dimensions framework, suggests that features that correspond with different concepts should have different names. When different features are given similar names, users may be confused; this problem can be seen in C's `const` syntax, where the position of `const` is significant but it is not obvious which position has which meaning. The differences between proposed features can be subtle on the surface even though the features represent significantly different meanings. For example, the system by Haack et al. includes qualifiers `RdWr`, `Rd`, and `Any` [88]. One might guess that `Any` is equivalent to `RdWr` because read and write would seem to be the only available kinds of access, but in fact `RdWr` corresponds to a mutable object, `Rd` corresponds to an immutable object, and `Any` corresponds to a read-only restricted reference. `Any` means that the actual immutability invariant is either `RdWr` or `Rd` and therefore a holder of a reference to an `Any`-object cannot write to it because it might be immutable (`Rd`). It is easy to imagine users confusing these different terms, based on the heuristic evaluation evidence; we lack stronger evidence since Haack and Poll did not publish a usability study of their system.

This problem of features with potentially confusing names is not limited to just one system. IGJ [223] supports an object annotation `@Immutable`. However, in order to make immutability transitive, one must use the immutability parameter on fields in the transitive closure of objects that are referenced by the `@Immutable` object. This means that when a programmer has a reference to an `@Immutable` object, it is necessary to locate and read an arbitrary amount of class implementation code to determine what immutability means for this particular object. Furthermore, a future code edit may invalidate the programmer's reasoning, resulting in the programmer making an immutability assumption that is later violated by a programmer who was unaware of the previous reasoning. This is a *hidden dependency* and may cause bugs. Systems that support removing fields from the abstract state, such as C++'s `mutable`, have a similar problem. However, the real-world implications of these design choices have not yet been tested.

JAC [109] provides a larger collection of features: `writable`, `readonly`, `readimmutable`, and `readnothing`. `Readimmutable` methods can only read the parts of an object's state that are immutable after initialization. The authors say that `readimmutable` may be useful for objects used as keys in hash tables, since the parts of those objects that contribute to the hash code must not change while the objects are used as keys. However, it is unclear whether this complexity is warranted; perhaps it would be better to simply require that the entire object be immutable. We view this question as a tradeoff between flexibility and simplicity rather than assuming that more flexibility is always better.

4.5 Interviews with Developers

4.5.1 Methodology

Given the large collection of immutability- and read-only restriction-related design choices that must be made in order to construct a concrete programming language or language extension, we wanted to find out how practicing software engineers think about state when writing software. We obtained IRB approval and conducted semi-structured interviews with a convenience sample (N=8) of software engineers at several US- and Europe-based organizations. We focused on software engineers who work on large software projects, with the assumption that these projects would be the most likely (relative to smaller projects) to encounter interesting problems with state management. Likewise, due to the participants' expertise, any problems raised are likely to come up in a wide variety of situations and be relevant problems for consideration in a language design. Our participants had spent a long time as software engineers, with a mean professional experience of fifteen years and a minimum of seven years. They typically had worked on projects with millions of lines of code and hundreds of people. Participants reported significant usage of immutable and access-restricted interfaces and they had a multitude of strong opinions about state management in large software systems. Their experience was primarily in C++, Java, and Objective-C. Each interview took between 45 minutes and an hour and a half. We recorded the interviews and took notes of the participants' answers to our questions; then, we informally aggregated the responses to find common themes and interesting anecdotes. Because we conducted a small set of interviews, our goal was to extract as much information as possible from the interviews rather than to gather statistically significant quantitative data from the responses. We stopped recruiting participants when we observed that most of the answers duplicated responses that we had already heard.

Interview Questions

1. How long have you been programming professionally?
2. Can you give an order of magnitude estimate of the size of the largest project you've made significant contributions on? Number of people, lines of code?
3. In what programming languages do you consider yourself proficient?
4. How did you get into software development? Do you have a computer science background?
5. Let's talk about changes that happen to state in software you've worked on. Many kinds of software maintain state, such as object graphs, files, or databases, but there's a possibility of corruption during changes due to bugs. How do you make sure that state in running programs remains valid?
   (a) Are there specific techniques do you use? If so, what are they?
   (b) Do you sometimes want to make sure that some operations don't change any state or don't change certain state?
      i. Tell me about a recent time you did this.
      ii. How often does this come up?
      iii. Do you use language features to help?

(c) Do you sometimes want to make sure that some state never changes?

(d) Tell me about a recent time you did this.

i. How often does this come up?

ii. Do you use language features to help?

iii. Do you sometimes want to make certain kinds of modifications to state impossible for some users of an API but not others? If so, how do you do that?

6. How often do you work on concurrent aspects of software? What mechanisms do you use to control concurrency?

   (a) Do you use immutability to help address or prevent concurrency issues?

7. How much work have you done on security-related aspects of your software? Have you found or fixed any vulnerabilities?

   (a) Do you use immutability to help address or prevent security issues?

8. Can you recall a recent situation in which you created an immutable class or other data? If so, tell me about it.

9. Can you recall a recent situation where you changed a class from immutable to mutable? If so, tell me about it.

10. Can you recall a recent situation in which you changed a class from mutable to immutable. If so, tell me about it.

11. Can you think of a bug you investigated or fixed that was caused by a data structure changing when it should not have? What was the problem and how did you solve it (if you solved it)?

    (a) Would const have prevented the bug?

12. Have you ever tried using an immutable class and had it not work? Why not?

13. When you create a new class, how do you decide whether instances should be immutable?

14. Have you ever been in a situation where you wanted to use const or final but it didn't say what you wanted to say?

    (a) or where you discovered you couldn't use it? What was the situation and why couldn't you use it?

15. Have you been in a situation where you had to revise your plan because something you'd assumed could mutate state was disallowed from doing so due to const?

16. Have you been involved in training new members of the team? What do you tell new members about immutability or ensuring invariants are maintained?

17. Sometimes, though an object is mutable after creation, it only needs to be changed for a short amount of time. For example, when creating a circular data structure, the cycle must be created after allocating all the elements. After that, however, the data structure doesn't need to be changed. Have you encountered situations like that? Do you think it would help if you could lock the object after all necessary changes were made?

18. Now, I'd like to move on to API design in general. Think of a recent API you designed.

(a) Did you make any conscious design or implementation decisions, to make the API easier or more manageable for these users?

(b) Are there any recurring issues / challenges that users have had with your API? How did you handle those?

(c) How do you differentiate between users of your API? Are there parts of the API that you expose some users but not others? How do you manage that?

(d) Did you make any conscious design or implementation decisions to protect key data or data structures from modification (inadvertent or malicious) from your users?

19. Thanks! Is there anything else you'd like to say?

4.5.2 Results

Relevance. Asked about bugs caused by state changing when it should not have, one participant exclaimed,

> "Oh God, like, most of them!...my favorite is where you have data that is supposed to be immutable and is only settable once in theory but that's not well enforced and so it ends up getting re-set later either because it gets re-initialized or because someone is doing something clever and re-using objects or you have aliasing where two objects reference the same other object by pointer and you make changes...".

Another participant cited library boundaries as a problem: this engineer's module depended on data that got changed by a third-party library, resulting in half-updated or not-updated state. This resulted in mistrust among the groups because it was frequently unclear which team was responsible for any given bug. All the participants who worked on software with significant amounts of state said that incorrect state change was a major source of bugs.

General techniques. All of the participants reported using various techniques to ensure that state remained valid. Techniques included ensuring that lifecycle and ownership are well-defined using conventions such as C++'s RAII; restricting access with private variables and interfaces; using consistency checkers and writing repair code; unit and manual testing; and assertions. According to one participant, good design is better than using `const`: "if you simply do not depend on an object, there is no way you could possibly modify it directly." Another participant uses immutability as a key part of the architecture: "By design, we've made the key data structures immutable. We never update or delete any data objects. This ensures that an error or problem can never put the system into an undesirable state."

C++ `const`. Several participants used `const` in C++ to make sure state does not change, but it does not meet their needs. First, there is no way to make special cases for fields on a per-method basis: either fields are part of an object's abstract state or not. One participant reported removing `const` from a collection of methods rather than making a particular field mutable. Second, the viral nature of `const` makes using it a significant burden, since declaring a method `const` requires transitively annotating all methods that the first method calls. One participant wished for tools that automatically added `const` as needed in such cases; for Java, Vakilian et al. proposed a tool that would help programmers add similar annotations [207].

One participant complained that `const` applies to methods and fields but not to whole classes. In contrast, another participant said that marking methods `const` is very helpful, as is having two interfaces to every object: a mutable interface and a const interface. The former approach corresponds to class immutability if all methods are marked `const`; the latter approach implements read-only restrictions if all methods in one interface are const.

Participants said that there were many situations in which they wanted to use `const` but it did not express the invariant they needed. One participant wanted to restrict access in a more fine-grained way than just mutability: for example, by disallowing access to particular methods even though those methods did not change state, or permitting some kinds of state changes but not others. Another participant cited the discrepancy between abstract and concrete state: though one can mark particular fields as `mutable` in C++, it would be preferable to be able to show that the value of a field has no effect on views of the object.

Thread safety. We were particularly interested in how participants used immutability to manage state that was shared across threads, since free sharing across threads is a frequently-cited benefit of immutable data structures. We asked participants what techniques they use to ensure state is safe when accessed concurrently. Two participants described architectural techniques that hide concurrency from users, avoiding this problem. Immutability did not seem to be a commonly used technique; participants cited traditional methods, such as serial queues and mutexes. One participant, a framework designer, mentioned a focus on minimizing dependencies on mutable state across threads. A major theme that arose was about reuse: when reusing existing structures, participants had to assume that structures were mutable because they had no guarantees otherwise. One participant mentioned taking advantage of immutability if it was already present. Another participant pointed out that it is rare to be able to design a component from the ground up, so there is usually some synchronization needed. This suggests that when designing components for reuse, it is helpful to be able to specify to consumers of the components which aspects are immutable. We conclude that existing techniques for specifying immutability in the languages these participants used are insufficient for facilitating reuse in concurrent systems.

Immutable classes. We asked participants about their use of immutable classes and found that immutable classes are used very frequently but that languages the participants use do not provide any explicit support for them. In fact, one participant pointed out that some languages make immutable classes difficult to write and use. In C++, copying objects is difficult and error-prone, so one company's style guide recommends disabling copy and assignment operations. As a result of the difficulty of copying objects, objects tend to be mutable.

Participants mentioned using immutable classes for copy-on-write situations; for objects that manage relationships between other objects; and for wrapping an existing mutable class in order to enforce a particular invariant. One participant mentioned avoiding changing classes from immutable to mutable because then anything holding a reference to the object would suddenly depend on whatever could now cause mutations. This again is a notion of class immutability, not object immutability. That is, instances of a class serve a particular role in an architecture, and the class defines the role; making all instances mutable would violate invariants of many of the clients of that class.

Security. Since many recommendations in favor of immutability come from the security community [180] [165], we asked participants about their experiences with security considerations in state management. One participant talked about a security-related object that the team wanted to

make immutable but which needed to be modified as part of the teardown process, so it had to be made mutable instead. None of our participants had a focus on security in their jobs, though most of them worked on software that could potentially have security problems. Their perspective was generally that security was important in security-related components, such as authentication, but otherwise not a primary concern. One participant, who worked in part on cloud-based software, mentioned that privacy is a more difficult issue than security because the requirements are less clear and because privacy issues come up in more contexts.

4.5.3 Implications on language design

From the above interview findings, we extracted the following observations that are relevant for the design of programming languages and tools. Since the interviews were of programmers working on large, object-oriented systems, our interview findings are most applicable to object-oriented languages that are intended to be used for large systems.

- Both read-only restrictions and immutability are used in practice for different purposes, but the languages that our participants used do not reflect their needs.
- State management is a core issue that developers consider when designing architectures and APIs, and developers do use read-only references and immutability (when available) to enforce encapsulation.
- Existing mechanisms do not solve the problems that developers have when building concurrent systems, in part because they frequently re-use existing code that does not provide immutability guarantees. Lack of transitivity of existing mechanisms results in guarantees that are too weak to be useful.
- Incorrect state change is a frequent cause of bugs. As a result, programming language features that help developers manage state have a good chance of preventing many bugs and so research about such features is a worthwhile endeavor.
- Even limited read-only restriction mechanisms, such as `const`, can be too hard for programmers to use effectively. Designs must emphasize simplicity and usability, or the features will not get used.
- Facilitating immutability is a core issue in programming language design; languages that make copying objects onerous or expensive discourage programmers from using immutability-related features.

4.5.4 Limitations

Our small sample size resulted in several limitations to our findings. We do not know whether programming languages used for small projects or for short-lived projects would have the same design considerations, since many of the justifications for the recommendations came from experiences with large, long-lived projects. It is unclear to what extent the participants' backgrounds biased our findings, since nearly all of them had formal computer science training, whereas across the industry, software engineers come from a variety of different backgrounds. Furthermore, our participants were very experienced; novices may benefit from different language design choices

than experts do.

4.6 A Pilot Study of Transitive Immutability Restrictions

Following techniques from the human-computer interaction community [136], we conducted a user-centered, iterative design process to create language features with which users can express state change constraints in a way that is both usable and useful. Rather than assuming that users need all possible features, we are starting from the assumption that the language should be as simple as possible while still providing what users need to address their problems. There is a tradeoff between simplicity and completeness: adding keywords for all possible features that users might need might make addressing more problems *feasible* but also make the system so complex that users cannot successfully use those features correctly. By designing our system according to the needs that practitioners have expressed and iteratively testing specific designs with users, we hope to arrive at a practical solution that addresses many, but likely not all, of the problems that practitioners face in this space.

We must distinguish between requests from practitioners and their actual needs. Users frequently do not know what features would benefit them most [16], so asking them what features they want is only the beginning of the process. We grounded our interviews by asking primarily about their experiences and only secondarily about their feature requests; by basing our designs on the requirements of the systems the users were building, we increase the chance that our system is effective on real-life problems.

4.6.1 Transitivity

Based on our finding that some programmers want guarantees that can only be provided by transitive immutability and the disparity in transitivity support between research languages and commonly-used languages, we focused on the question of transitivity. We piloted a user study comparing a non-transitive subset of IGJ [223] with a modified version that always enforces transitivity, which we call IGJ-T. We started with IGJ due to its availability and practicality: it can be used by experienced Java programmers and requires learning only how IGJ annotations are processed, not a whole new programming language. Initially IGJ was an extension to Java using generics [223], but we used a revised version based on the Java annotation system [167].

Because we wanted to compare always-non-transitive immutability to always-transitive immutability in our study, we only told our participants about the `@Immutable` annotation, not any of the others. This forced a non-transitive approach to immutability for the participants in the control condition. For the transitive case, which we wanted to be identical to IGJ except for transitivity, we created a version of IGJ, *IGJ-T*. In IGJ-T, if a class has a constructor that returns an `@Immutable` result, then all fields of that class must be transitively `@Immutable`. The implementation of IGJ-T works by visiting all methods when typechecking; for methods that are constructors that are annotated `@Immutable`, it recursively visits the types of the class's fields and verifies that all of them are marked `@Immutable`.

The design of IGJ greatly facilitated verification of transitive immutability relative to writing an analyzer from scratch. IGJ is implemented as part of the Checker framework [168], which

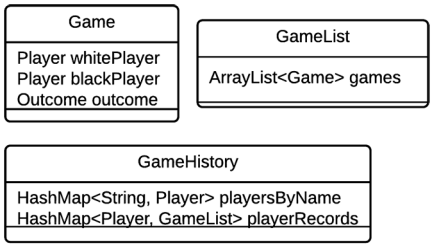

Figure 4.1: Class diagram of starter code

provides a collection of related facilities for checking properties with annotations in Java. In particular, the Checker framework made it very easy to extend IGJ with a new verification on constructors.

IGJ-T gives error messages like the following example:

```
error: [transitivity.invalid] Cannot declare
Simple() with annotation @Immutable because Simple
transitively contains @Mutable Date d on path
@Immutable AContainer a -> @Mutable Date d
    @Immutable Simple () {
               ^
```

IGJ-T gives the full field path to the erroneous field so that programmers can easily find the cause of the problem.

4.6.2 Study design and pilot results

We designed four tasks, intended to take 90 minutes total, in the context of a program to track outcomes of chess games (Figure 4.1 summarizes the architecture). After obtaining IRB approval, we recruited a convenience sample (N=3) of PhD students experienced in Java to pilot our study. Two participants completed a pre-test regarding their programming experience and understanding of `final`. Of these, one participant with over a year of Java experience, on hearing an explanation of the fact that `final` only applies to assignment but not to the referenced objects, exclaimed, "no one ever highlighted that key thing [before]!" The tasks are summarized below:

1. Suppose your language has a feature that lets you specify that once a Game has happened, it should not be changed. Please make any changes necessary to the existing program to express this using whatever language feature you think would be best.
2. Now, we will show you the feature we have designed, which we call `@Immutable`. (Participants were then given a description of IGJ or IGJ-T depending on the experimental condition to which they were assigned). For the following sample code, please write which lines would give compile errors to test your understanding of `@Immutable`. (The sample code included a Person class with fields for eye color and name; the eye color was supposed to be fixed and the name could be changed. The instructions showed how `@Immutable` could be used to make the eye color immutable and compared `@Immutable` with `final`,

which would not suffice when used only on the eye color field because the eye color was a reference to a Color object.)

3. Please implement the `updatePlayerName()` method to change the name of a player.
4. Please implement the `changeGameOutcome()` method to update the history for a corrected game outcome. Game must remain `@Immutable`.

The first task was designed to elicit how the participants would want to express the immutability concepts, if given free reign, using the natural programming elicitation approach. Note that the prompt (and the instructions beforehand) did not use the word *immutable* in order to try to avoid biasing the participants' vocabulary choice.

The second task introduced the design of the immutability construct that participants would be using and verified that they understood it in a small test. All of the participants understood their version of `@Immutable`.

In the third task, we expected participants in the IGJ condition to erroneously mutate an object that was used as a key in a hash table, resulting in a bug; we expected participants in the IGJ-T condition to spend longer on the task but avoid the bug. In the fourth task, we expected participants in the IGJ-T condition to spend extra time solving the problem but reap little benefit.

Though the user study was very small-scale (three participants, and the first was only given the first two tasks), we have learned helpful insights about Java programmers' expectations. In the first task, one participant used `immutable` as a qualifier on the Game class declaration; the other added a `freeze` qualifier on a Game parameter to one of the constructors of GameList (neither participant used the Java annotation syntax).

The third task required participants in the IGJ-T case to essentially rewrite the entire game history; in the IGJ case, a simpler rewriting was possible. In both cases, however, the participants were surprised that they had to rewrite data structures rather than modifying them in place: "It seems complicated because even if you want to change the name you have to reconstruct everything." Some Java programmers appear to find immutable data structures confusing, and encountering them forces a difficult problem-solving process.

One participant in the non-transitive case had recently been doing more functional programming, so we expected that this experience might make immutability more intuitive. However, this participant modified Person in place, not realizing that this would break various aspects of the data structures, in part because Person was being used as a hash table key. When this participant discovered the problem while testing and debugging, the participant exclaimed, "this is what happens when you mutate [stuff] in place!" and commented that maybe upon switching back to an imperative programming context, it was hard to remember all the problems that are inherent in imperative programming.

Due to the time spent on the third task, one participant did not start the fourth task; the other two found it similar to the third task.

Though we restricted our study to a small subset of IGJ, some participants had difficulty understanding its error messages while trying to fix their bugs. For example, one participant spent over two minutes debugging this error message:

```
error: [method.invocation.invalid] call to
setBlackPlayer((@org.checkerframework.checker.igj.
qual.Immutable :: t_starter.Player)) not allowed
```

```
on the given receiver.
    game.setBlackPlayer(newPlayer);
                        ^
found   : @Immutable Game
required: @Mutable Game
```

The participant kept checking the argument to `setBlackPlayer`, but the problem was that `game` was `@Immutable` and the solution was to create a new `Game` object instead of modifying the existing one. Note that this error was an existing part of IGJ. In our limited pilot, only one user used IGJ-T, and that user did not encounter any transitivity error messages.

Threats to validity include the small set of tasks we used in comparison to the wide variety of tasks that real software engineers perform; the small codebase and short timeframe of the study; and the relative inexperience of our participants, who were mostly graduate students with small amounts of professional experience.

4.7 Conclusion

Despite the vast design space for language features supporting immutability in programming languages and plentiful advice regarding how programmers should use them, there is only scarce empirical evidence supporting these recommendations. We presented a classification of immutability features, an analysis of their tradeoffs, and a pilot user study, challenging the notion that providing a very flexible feature set is best. In the next chapter, we will describe Glacier, which arose from what we learned designing and evaluating IGJ-T. By focusing on a simple set of features, we were able to achieve our usability objectives while maintaining the expressiveness needed in many common cases. Chapter 5 continues the discussion of immutability by describing Glacier, a system we designed to address some of the usability difficulties we observed in IGJ-T while still meeting our design goals.

Chapter 5

Glacier: Transitive Class Immutability for Java[1]

5.1 Introduction

The space of immutability support in languages is complex; chapter 4 describes eight dimensions along which a language can support immutability. If a programming language is to support the specification and enforcement of immutability, what kinds of immutability should the language support? Supporting as many different kinds of immutability as possible results in a complex system; to date, there are no usability studies published of immutability specification systems. Our studies of IGJ-T in chapter 4 showed how supporting different kinds of immutability at once can result in a system that is very difficult to use effectively and correctly. Alternatively, a design that supports a small set of immutability-related features might be easy to understand and apply but fail to capture useful constraints. Such a system might fail to achieve the goals of immutability systems: preventing bugs and documenting and enforcing specifications. This motivates the research question for this chapter: can we select a subset of immutability features and design a corresponding programming language such that:

1. Real users can use the immutability restrictions effectively with minimal training; and
2. Expressing immutability with the language actually prevents bugs in situations where software engineers have already decided on an immutable design?

To address these questions, we designed, implemented, and evaluated *Glacier*, a type annotation system for Java. *Type annotations* are an existing mechanism in Java that supports extending the type system. Based on the observation that programmers would benefit from strong guarantees (chapter 4), we focused on *transitive class immutability*. *Transitivity* ensures that immutable objects can never refer to mutable objects; *class* immutability means that immutability of an object is specified in its class's declaration. Glacier, which stands for *Great Languages Allow Class Immutability Enforced Readily*, enforces immutability statically, with no effect on the runtime and therefore no performance cost on the compiled software, so that users can get strong guarantees at compile time.

[1] Portions of this chapter previously appeared as [46].

We evaluated the practicality, applicability, usability, and usefulness of Glacier in two case studies on existing code and in a user study with 20 participants. In the case studies, we successfully applied Glacier to a spreadsheet model component and to a reusable immutable container class, observing that Glacier is applicable to these real-world, existing software systems. In applying Glacier, we also found two previously-unknown bugs in the spreadsheet implementation. In the user study, we compared Glacier with `final`, since `final` is the current state-of-the-practice mechanism for specifying immutability in Java. When given programming tasks, users in the condition where they only had `final` all made various errors that resulted in breaches of immutability, even after receiving explicit training in how to use `final` correctly; in contrast, although the participants who used Glacier had never seen it before, almost all of them succeeded in using it to specify immutability correctly. We also asked participants to complete programming tasks with immutable classes, and found that although most users were able to complete the tasks, all users of `final` wrote code that had bugs or security vulnerabilities due to improper mutation; Glacier prevented these problems at compile time.

This chapter makes the following contributions:

1. A definition and formal model of transitive class immutability as an extension to Featherweight Java [100];
2. An implementation of that model in a tool called *Glacier*, which enforces transitive class immutability in Java. By enforcing only the kind of immutability for which there is the strongest empirical support, we have achieved significant simplifications relative to existing systems;
3. Evaluations of Glacier in two case studies on real software projects that showed that Glacier captures a kind of immutability appropriate for those projects;
4. The first formal user study of any immutability system. We compared Glacier to `final` and found that all ten participants who used `final` wrote code that had bugs or security vulnerabilities, even after having been trained on correct `final` usage, in a situation in which Glacier statically detects those problems. Almost all the Glacier users were able to complete the tasks successfully.

Designing an enforcement system requires making a collection of design choices regarding what immutability means and how it will be enforced. We identified eight distinct dimensions of immutability chapter 4, resulting in at least 512 different combinations of features. As such, any proposal should include a justification for its position in the design space.

Design recommendations. Chapter 4 described our interviews with professional software engineers and our conclusion that the *transitive immutability* subspace seemed to reflect the needs of our interviewees. Immutability can provide particularly useful guarantees: immutability provides guarantees regarding state change, rather than guarantees regarding access (as in the case of read-only restrictions). Relative to non-transitive immutability, *transitive* immutability is more useful: the entire state of an immutable object is immutable, rather than just a part that depends on the object's implementation. For example, if a *transitively* immutable `Person` object has a reference to an `Address` object, `Address` must be immutable as well. As a result, objects that are transitively immutable can be shared safely among threads without synchronization, and invariants that are established regarding objects' state are always maintained. Our interviews also found evidence in support of *class* immutability, with some engineers observing that most classes

serve a particular architectural role, and that role typically either requires mutability or not.

5.2 The Design of Glacier

5.2.1 Evidence-based design

We designed Glacier using an evidence-based approach. Based on our prior findings showing that transitive immutability provides particularly useful guarantees, we concluded that Glacier would support transitive immutability. In order to facilitate practical usage of Glacier, since Witschey et al. found that simplicity and ease of use are predictive of adoption [218], we designed Glacier to be as simple as possible while still enforcing immutability. Glacier is a static typechecker, so it provides strong, compile-time guarantees and imparts no runtime cost on programs. When invoking the existing Java compiler on the command line, users can pass a command-line argument that causes the compiler to invoke the Glacier annotation processor; users of a build system can arrange to always pass this argument by default. This approach has practical advantages, since teams can adopt Glacier without changing compilers and individual programmers can choose when to invoke the checker, for example skipping checking to temporarily use unsafe debugging code. However, it is possible to circumvent these checks by not running the annotation processor. For example, from a class that is compiled without Glacier, one could modify a public field in an `@Immutable` class defined in an external .jar file.

Favoring simplicity over expressiveness may limit adoption by limiting applicability. For example, fields in immutable objects cannot be initialized lazily or include circular references.

Figure 5.1 shows several design alternatives, comparing Glacier to the earlier IGJ and IGJ-T. #1 shows how in IGJ, immutability is a property of references, not of classes. Compared with IGJ, IGJ-T (#2) also enforces that fields of classes for which there are immutable instances must have all-immutable fields (enforcing transitivity). In Glacier (#3), immutability is a property of classes, not of references. Variant #4 explores a possible extension of Glacier, in which the compiler can automatically derive immutable versions of mutable classes. We elected not to pursue that approach, since our interview had indicated that most classes are used either in an immutable or a mutable way, but not both.

5.2.2 Syntax and context

We were interested in evaluating our tool in the context of an existing corpus of code and with programmers who might be able to use it. As such, we implemented Glacier in the context of Java, which has a large and active user base. Java is also representative of a broad class of object-oriented languages. Implementing Glacier as a type annotation processor has several benefits over a from-scratch approach: by using Java type annotations, Glacier uses only existing Java syntax and can be parsed by the standard Java parser. Glacier is implemented within the Checker Framework [168], which facilitated Glacier's development. In Glacier, types can be annotated with `@Immutable` to indicate that they are immutable. Types that are not annotated are not guaranteed to be immutable. Glacier represents this with an implicit annotation of `@MaybeMutable`. Here is an example of a simple `@Immutable` class:

1: IGJ. Immutability is a property of references. Immutability is not transitive.

```
public class Game  {
  // No need for 'outcome' to be immutable, since immutability is not transitive.
  private Outcome outcome;

  // Constructor returns a reference to an immutable object.
  @Immutable Game (Outcome outcome) {
    this.outcome = outcome;
  }
}
```

2: IGJ-T. Immutability is a property of references. Immutability is transitive.

```
public class Game {
  // @Immutable is required when declaring 'outcome' because there is
  // at least one constructor that returns an @Immutable reference.
  private @Immutable Outcome outcome;

  // Constructor returns a reference to an immutable object.
  @Immutable Game (@Immutable Outcome outcome) {
    this.outcome = outcome;
  }
}
```

3: Glacier. Immutability is a property of classes. Immutability is transitive.

```
@Immutable public class Game {
  private Outcome outcome; // OK because Outcome class was declared @Immutable

  // Every instance of Game is immutable, so no need to specify immutability.
  Game (Outcome outcome) {
    this.outcome = outcome;
  }
}
```

4: An variant of Glacier, in which the compiler can synthesize immutable subsets of mutable classes.

```
// Assume Outcome is mutable, but the compiler can synthesize an
// immutable version by leaving out the mutating methods
@Immutable public class Game {
  Outcome outcome; // error: Outcome is mutable
  @Immutable private Outcome immutOutcome; // OK: use immutable subset

  Game (Outcome outcome) {
    // Assignment is always permitted in the constructor
    this.outcome = outcome;
  }

  void test () {
    this.immutOutcome.setOutcome(WON); // compile error: no such method
    this.outcome = ... // compile error because Game is @Immutable
  }
}
```

Figure 5.1: Alternatives we considered for immutability systems.

```
@Immutable class Person { ... }
```

5.2.3 Class immutability

Some systems, such as IGJ [223], support both class immutability *and* object immutability. However, we seek to design a system that is as simple as possible and yet still reflects users' needs. We found, perhaps surprisingly, that supporting only class immutability and not object immutability resulted in significant simplifications to the system. For example, suppose `@Immutable` could be applied to objects and classes. Consider an identity method:

```
interface DateUtilities {
  public Date identity(Date d);
}
```

The declaration of `identity` does not specify whether its argument is `@Immutable`. A caller of `identity` may require that the annotation on the returned object is the same as the annotation on the passed object, but the interface does not provide that guarantee. Polymorphism addresses this problem:

```
interface PolymorphicDateUtilities {
  public  @I Date identity(@I Date d);
}
```

This notation means that the `Date` input to `identity` has some annotation, `@I`, and the returned object has the same annotation. Though polymorphism increases flexibility, adding this feature increases the complexity of the language.

Another problem with object immutability pertains to the subtyping relationship between mutable and immutable instances of a particular class. Consider a method that took an `@Immutable` object as a parameter. Passing a `@MaybeMutable` object would be unsafe because the method might assume that no state in the object will change in the future, perhaps sharing it among threads. Likewise, a method that expects a mutable object cannot take an immutable object because the immutable object lacks mutating methods. Therefore, in Glacier there is no direct subtyping relationship between the mutable and immutable types. By supporting only class immutability, the user can decide whether each class should be mutable or immutable, and then there is no question of subtyping among objects of the same class. Alternatively, one could have a common supertype of both the immutable and mutable subclasses. This requires introducing a third type, again resulting in more complexity. Either the user has to manually implement all three classes, or there must be a system by which the user may specify how to generate them.

Supporting only class immutability also simplifies error messages: on seeing an error message pertaining to an annotated type, the user can always know that the annotation came from the class's declaration or a conflicting local annotation, rather than via type inference (which would otherwise be important to avoid the proliferation of annotations on all types). When a user sees a type name, the annotation is implicit; if there is a declaration of `@MaybeMutable class Date`, then there is no need to annotate any other usage of the `Date` type because every `Date` is `@MaybeMutable`. Likewise, an object is immutable if and only if its class is immutable, simplifying reasoning. These semantics are formalized in § 5.2.9.

5.2.4 Restrictions of immutability

Glacier enforces two restrictions on the fields of `@Immutable` classes: all fields must be `@Immutable`, and fields cannot be assigned outside the class's constructors. Note that the former requirement implements *transitive* immutability: an `@Immutable` class's fields must all be `@Immutable`, so the referenced objects cannot have their fields reassigned or refer to mutable objects, etc. `final` is permitted on fields but is redundant with `@Immutable` on the containing class. For example:

```
@Immutable class Person {
  String name; // OK; String is @Immutable
  Date birthdate; // Error; Date is @MaybeMutable

  void setName(String n) {
    name = n; // Error; Person is @Immutable
  }
}
```

When a reference to an object is of `@Immutable` type, Glacier guarantees that the referenced object is immutable. However, if a reference type is not `@Immutable`, Glacier provides no immutability guarantees. In particular, the referenced object may dynamically be `@Immutable`. As a result, subclasses of `@Immutable` classes must be `@Immutable`, but subclasses of `@MaybeMutable` classes can be either `@MaybeMutable` or `@Immutable`. Importantly, a subclass of a `@MaybeMutable` class can only be `@Immutable` if no superclass has a non-final field or field of `@MaybeMutable` type. Likewise, if an interface is declared `@Immutable`, then all implementing classes must be `@Immutable`, but a `@MaybeMutable` interface can be implemented by an `@Immutable` class. All subinterfaces of `@Immutable` interfaces must also be `@Immutable`. This ensures that all subtypes of an `@Immutable` type are `@Immutable`.

It is not obvious that it should be permitted to declare an immutable subclass of a mutable class. It might seem that if the superclass has a guarantee of mutability, the subclass should adhere to that guarantee. However, that is precisely why the alternative to `@Immutable` is `@MaybeMutable`: a `@MaybeMutable` class is *not guaranteed to be mutable*. A significant disadvantage of this design decision is that adding a non-final or `@MaybeMutable` field to a `@MaybeMutable` class is a breaking change for `@Immutable` subclasses; this disadvantage is compounded by the fact that subclasses may not even be in the same package as the superclass and the implementor of the superclass may not be aware of the existence of subclasses. However, the problem of changes in superclasses unexpectedly breaking subclasses is long-standing in object-oriented systems and is well-known as the *fragile base class problem* [127]. Enabling immutable classes to subclass certain mutable classes enables existing, commonly-used design patterns to be compatible with Glacier. For example, Google's Guava libraries [162] provide an ImmutableList class that (indirectly) extends java.util.AbstractCollection, which cannot be `@Immutable` because it has mutable subclasses. However, ImmutableList itself does not support mutation and can be annotated `@Immutable`. Because practicality is a design objective of Glacier, we considered allowing `@Immutable` subclasses of `@MaybeMutable` classes a good tradeoff to make. It might be possible to address the fragile base class problem by supporting

an additional annotation for classes that must not refer to any mutable state but which may have `@MaybeMutable` subclasses.

Java primitives, such as `int`, are `@Immutable`; assignment to a primitive-type variable reflects binding the variable to a different primitive, not a mutation of an existing value. Glacier includes a list of JDK classes, such as `String` and `Integer`, that are `@Immutable`, but that list does not currently include all immutable classes in the JDK.

It is an error in Glacier to give an annotation to a type use that is different from annotation given at the type's declaration. If no annotation is provided in the type's declaration, then the annotation `@MaybeMutable` is implicit. As a special exception, both `@Immutable Object` and `@MaybeMutable Object` are permitted, so that all `@Immutable` types have a common supertype that specifies immutability. For example, one can specify a container that can hold any immutable object. `@Immutable Object` is a subtype of `@MaybeMutable Object`: a `@MaybeMutable Object` can refer to any object at all. For example:

```
@Immutable class ImmutableContainer
   <T extends @Immutable Object> { ... }
class Container<T> { ... }
```

In the Checker framework, receiver annotations are contravariant with respect to overrides: an overridden method must be called on an object with a supertype annotation of the original method's receiver. This ensures that if the method was invokable on an object of superclass type, it will be invokable on an object of subclass type as well. Glacier overrides this to permit covariant annotation overriding in the receiver. This allows methods of Object to be overridden in `@Immutable` subclasses, and it is safe because dispatch to the method of an `@Immutable` subclass implies that the `@Immutable` annotation on the receiver is correct.

5.2.5 Additional annotations for arrays

Arrays are an older Java feature and their design has various inconsistencies with other aspects of Java, so they pose some special problems. For example, occasionally it is desirable to write a method that can take both mutable and immutable arrays; this is safe if the method can be statically guaranteed to never reassign any of the array elements. Note that this case does not arise with other kinds of objects because with other objects, the types dictate the immutability annotation. To address this case, Glacier includes an additional annotation, `@ReadOnly`, which is used on array parameters to methods. A `@ReadOnly` array can also be referenced by a field that has a `@ReadOnly` array type.

The empty array poses a special problem: is it mutable or immutable? It is fundamentally immutable because it has no indices that can be modified, but it is also possible to declare a mutable array and reference an empty array with it. One workaround might be to declare two different empty arrays: one mutable and one immutable. Instead, the Glacier type hierarchy includes a *bottom element*, so named because it is a subtype of all other types. The bottom element, `@GlacierBottom`, applies to objects that have all properties of mutable and immutable objects, and can therefore be used when one wants either kind of object. One can declare an empty array of Object as follows: `static final Object @GlacierBottom [] EMPTY_ARRAY =`

`new Object @GlacierBottom [0];`. `null` also has annotation `@GlacierBottom` because it can be assigned to references with any annotation.

When any new object is allocated, it is guaranteed to not be aliased directly. For example, when the `clone()` method is called on an array, there are no aliases to that array (though there may be aliases to its elements). As a result, the result of a `clone()` call may be assigned to an `@Immutable` array or to a `@MaybeMutable` array; Glacier achieves this by annotating the return type of `clone()` with `@GlacierBottom`. Certain JDK methods also return `@GlacierBottom` arrays, such as `Arrays.copyOf`.

Our experience is that most users do not use arrays regularly, instead preferring collection classes, so the complexity of arrays may not be a significant burden to most users.

5.2.6 Typecasts

Normally, Java permits unsafe downcasts at compile time and checks for safety at runtime. Glacier has no runtime component, so unsafe casts are forbidden. For example, if `u` is of type `@Immutable C`, then Glacier reports a compile error on this cast: `((@MaybeMutable C)u)`.

5.2.7 Type parameters

Suppose an `@Immutable` class has a type parameter:

```
@Immutable class Box<T> {
  T obj;
}
```

If `Box` is instantiated with a mutable type for `T`, then `Box` contains a mutable object, which is a violation of transitive immutability. As a result, Glacier restricts type parameter instantiations on `@Immutable` classes to `@Immutable` types. This is a conservative approximation, since the type parameter may never be used in a field. However, checking whether a type parameter is used as a field depends on the implementation of the referenced class, which might result in confusing errors and which would violate modularity: if an immutable class was changed from not using its type parameter in a field to doing so, that would be a breaking change for clients that instantiated the class with a mutable type parameter. Furthermore, it is our experience that most generic classes use their type parameters in fields, so the conservative nature of this restriction is unlikely to be important in many use cases.

5.2.8 Robustness to future changes

One of the problems with `final` is that although it restricts assignability on fields to which it is applied, there is no way to specify that all fields of a class are `final`. When adding a new field to an immutable class, the author may neglect to mark the field `final`. Likewise, `final` cannot specify restrictions at the *usage* of a type, so clients of a class cannot ensure that it is `final`. Glacier solves this problem by permitting users to annotate any type use with an annotation. If that annotation is inconsistent with the annotation used in the type's declaration, Glacier will

Mod	::=	`assignable` | `final`
CL	::=	[`@Immutable`] `class C extends C implements` $\overline{\text{IF}}$ { $\overline{\text{Mod}}$ $\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$}
IF	::=	[`@Immutable`] `interface` I `extends` $\overline{\text{I}}$ {M-Decl}
K	::=	C($\overline{\text{C}}$ $\overline{\text{f}}$) { `super` ($\overline{\text{f}}$); `this.`$\overline{\text{f}}$ = $\overline{\text{f}}$; }
M-Decl	::=	C m($\overline{\text{C}}$ $\overline{\text{x}}$)
M	::=	M-Decl { `return` e; }
e	::=	x | e.f | e.m($\overline{\text{e}}$) | `new` C($\overline{\text{e}}$) | (C) e | e.f = e

Figure 5.2: Syntax of Glacier; gray boxes show differences with Featherweight Java.

report an error. This lets programmers specify that they depend on the immutability of a particular class they are using so that the compiler will report an error if that class is ever edited to make it mutable in the future. Although this offers an opportunity for authors of APIs to break clients, we think of this as exposing an *existing* mechanism of client breakage, which cannot currently be identified by the compiler.

5.2.9 Glacier formalization

The syntax of Glacier is shown in Figure 5.2. To help inform the design of Glacier, we also created a formal model (shown in Figure 5.3). Our formalism is an extension of Featherweight Java [100], which is a commonly-used minimal core calculus for Java. Gray boxes show changes in Glacier. For conciseness, not all rules from Featherweight Java are presented; those not presented are still part of the system. For example, casting is as in FJ.

5.3 Evaluation: case studies

5.3.1 Objectives

The restrictions that Glacier enforces were justified by the work described in chapter 4, our goals of simplicity, and the recommendations of experts [23], but do those restrictions reflect situations that arise in real software? Can Glacier work in software systems that are large and complex? Through case studies, we explored how applicable Glacier is to some kinds of real-world software systems. Though we cannot infer from case studies that Glacier is applicable to *all* systems (indeed, it likely is not), the goal of case studies was to gain an understanding of situations to which Glacier does apply and to refine the design of Glacier itself. For example, we found in the second case study that some immutable classes derive from classes that also have mutable subclasses; a previous formulation of Glacier did not reflect that use case. The case studies also drew our attention to the problems of overriding methods defined in `Object`. Finally, the case study systems provided a source of interesting test cases and helped us make Glacier more robust, particularly in the area of type parameters.

We sought case studies that would help us explore two kinds of use cases for Glacier. First, would Glacier be applicable to a large codebase corresponding to application software, given the

Syntactic MUTABLE and IMMUTABLE judgments: If an `@Immutable` class includes mutable fields, it will be judged IMMUTABLE but fail to typecheck.

$$\frac{}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \overline{\texttt{C}}\ \overline{\texttt{f}}, K\ \overline{M}\}\ \text{MUTABLE}}$$

$$\frac{}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \overline{\texttt{C}}\ \overline{\texttt{f}}, K\ \overline{M}\}\ \text{IMMUTABLE}}$$

$$\frac{}{\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\{\text{M-Decl}\}\ \text{MUTABLE}}$$

$$\frac{}{\texttt{@Immutable interface}\ I\ \texttt{extends}\ \overline{I}\{\text{M-Decl}\}\ \text{IMMUTABLE}}$$

MUT-FREE judgment: If a class's fields, including all fields introduced by superclasses, are all final and immutable, then the class may be used as a superclass of an immutable class. This is different from the IMMUTABLE judgment, which requires a declaration of `@Immutable`. The MUT-FREE judgment is used only to specify the static semantics.

$$\frac{}{\text{Object MUT-FREE}} \qquad \frac{}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\ldots\}\text{MUT-FREE}}$$

$$\frac{D\ \text{MUT-FREE} \qquad \forall i.A_i = \texttt{final} \wedge C_i\ \text{IMMUTABLE}}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \overline{\texttt{C}}\ \overline{\texttt{f}}, K\ \overline{M}\}\ \text{MUT-FREE}}$$

Subtyping:

$$\frac{}{\texttt{@Immutable Object <: Object}} \qquad \frac{[\texttt{@Immutable}]\ \texttt{interface}\ I\ \texttt{extends}\ \overline{J}\{\ldots\}}{I <: J_i}$$

$$\frac{[\texttt{@Immutable}]\ \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\{\ldots\}}{C <: D}$$

$$\frac{[\texttt{@Immutable}]\ \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\{\ldots\}}{C <: I_i}$$

Static Semantics: To the FJ rule for classes, Glacier adds the condition that D is mutable. Mutable classes cannot have immutable superclasses, since permitting that would allow a reference of immutable superclass type to refer to a mutable object.

$$\frac{\begin{array}{c}\texttt{fields}(D) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \text{OK in}\ \texttt{C} \qquad K = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\text{super}(\overline{\texttt{g}}); \text{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \\ D\ \text{MUT-FREE} \qquad \forall i.C_i\ \text{IMMUTABLE} \qquad \text{methods}(I) \subset (\text{decl}(\overline{M}) \cup \text{methods}(D))\end{array}}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \overline{\texttt{C}}\ \overline{\texttt{f}}, K\ \overline{M}\}\ \text{OK}}$$

$$\frac{\begin{array}{c}\texttt{fields}(D) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \text{OK in}\ \texttt{C} \qquad K = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\text{super}(\overline{\texttt{g}}); \text{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \\ D\ \text{MUTABLE} \qquad \forall i.I_i\ \text{MUTABLE} \qquad \text{methods}(I) \subset (\text{decl}(\overline{M}) \cup \text{methods}(D))\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \overline{\texttt{C}}\ \overline{\texttt{f}}, K\ \overline{M}\}\ \text{OK}}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad \texttt{fields}(C_0) = \overline{\texttt{C}\ \texttt{f}} \qquad \Gamma \vdash t_2 : C_i' \qquad C_i' <: C_i \qquad C_0\ \text{MUTABLE}}{\Gamma \vdash t_0.f_i = t_2 : \text{Unit}}$$

Figure 5.3: Formalization of Glacier semantics.

complex interactions among modules? Second, since generics pose special challenges, how would Glacier work in a library that made extensive use of generics? We selected ZK Spreadsheet, a Java-based spreadsheet implementation, to address the first question, and we selected the Guava ImmutableList class to address the second question.

5.3.2 Case study: ZK Spreadsheet Model

ZK Spreadsheet is a commercial, partly open-source, Java spreadsheet implementation [166]. It supports importing documents from Excel and provides a server-based spreadsheet component that can be inserted into web pages via an Ajax client-side component. As a case study of Glacier, we refactored the model portion of ZK Spreadsheet 3.8.3 (comprising about 36 KLOC) so that cell styles were immutable (cell styles record information required for correct visual rendering of cells, such as background color, font, etc.). We also updated the rest of the spreadsheet implementation (comprising about 21 KLOC) to use the new model. We added annotations so that Glacier could enforce immutability statically. The refactoring took me approximately 20 hours, not counting time spent fixing bugs in Glacier; this would likely have been less if we had already been familiar with the ZK codebase. In the process, we identified two previously unknown bugs in the spreadsheet implementation, one of which was due to incorrect copying code; in our revised version, no copying was necessary because immutable objects can be shared safely. The other bug related to font cache misses when changing fonts in cells.

Before starting, we asked the authors of ZK Spreadsheet whether they used any immutable structures, and they explained that they did not because they were wary of the performance cost of copying that would be likely if objects were immutable. However, cell styles can be shared among many cells, and ZK Spreadsheet has no data structure tracking which cells use a given style. Thus, to modify a cell's style, the system must make a fresh style, since modifying the existing style might incorrectly affect other cells. As a result, though the cell class was mutable, it was copied on nearly every modification. We believe the performance cost of using immutable styles is minimal; in fact, immutable styles may increase performance by facilitating safe sharing.

Our refactoring primarily used three strategies to convert mutable classes to immutable ones. In most cases, clients that mutated classes changed a small number of parameters at a time; in these cases, we added a new constructor that took the previous instance and the new value of the parameter. This approach was similar to that used by Kjolstad et al in their automatic refactoring tool [108]. Other classes had many attributes that typically needed to be modified at once; if those constituted most of the state of the object, the client called a constructor; otherwise, we used a mutable Builder object [77] to represent the collection of changes. This approach prevented overly verbose, inefficient implementations that would have resulted from using the first approach alone.

From our case study, we conclude that Glacier can be adopted to express and enforce transitive class immutability in some complex, real-world systems with a practical amount of effort.

5.3.3 Case study: Guava ImmutableList

After our initial case study on application software, we wanted to see how Glacier might be used on a very different system. Google's Guava project includes several immutable collection classes, including `ImmutableList`; though relatively small, this library is designed

to be used in a wide range of projects. We annotated `ImmutableList` and its superclass, `ImmutableCollection`, with `@Immutable` and made the appropriate changes necessary to make them compile. As a result of the use of generics in `ImmutableList`, when using Glacier, it was necessary to specify annotations for the bounds of the type parameters. For example, the original declaration of `ImmutableList` included: `public abstract class ImmutableList<E> extends ImmutableCollection<E>`. With Glacier, however, `E` is restricted to immutable objects, so the new declaration reads `@Immutable public abstract class ImmutableList<E extends @Immutable Object> extends ImmutableCollection<E>`. This constrains `E` to descending from `@Immutable Object`, expressing an *upper bound* on the type parameter. One might expect to write `@Immutable E` rather than `E extends @Immutable Object`, but Java specifies that `@Immutable E` expresses a *lower* bound on `E` rather than an upper bound; that is, it specifies that `E` must be a *supertype* of `@Immutable Object`, not a *subtype*.

ImmutableList included this method:
`static Object[] checkElementsNotNull(Object... array)`.
`checkElementsNotNull` took and returned a mutable array, but callers passed an immutable array to `checkElementsNotNull`, which was an error. Because Java methods cannot be overloaded with different annotations, we were unable to provide an alternative method with the same name that takes and returns an immutable array. This is one case in which polymorphism might be desirable. However, because `checkElementsNotNull` never modifies the input array, it is not actually necessary to return an array. We addressed this problem by refactoring this method to only do the checking and not return the input array.

The only aspect of ImmutableList that we were unable to represent in Glacier is a cache in ImmutableCollection, which caches an ImmutableList representation of the collection. Some languages, such as C++, permit exclusion of specific fields from enforcement of immutability. Glacier has no provision for allowing mutable fields in immutable objects so that Glacier can provide strong guarantees. A workaround would be to populate the cache inside the ImmutableCollection constructor, but this would have a performance cost if the list representation is never needed. Future work might explore extending Glacier to permit lazy initialization of fields in immutable objects; such initialization could be done safely if it is based only on state that was available at initialization time.

5.4 User study

Our user study of Glacier was designed to see whether Java programmers could use Glacier effectively with little training. We found that they could; in contrast, Java programmers without Glacier were unable to use `final` to express immutability correctly even after receiving appropriate training. We also found that Java programmers without Glacier wrote code that mutated immutable state, creating bugs and security flaws; Glacier detects these errors statically. Though there is a wide variety of proposals in the literature for systems that support immutability, we have not found any others that have been evaluated in a formal user study.

5.4.1 Methodology

We recruited 20 experienced Java programmers to participate in our study, which was approved by our IRB. For each sequential pair of participants, we randomly assigned one to a control condition, in which the participant used `final`, and the other to a treatment condition, in which the participant used Glacier, resulting in ten participants in each condition. After obtaining informed consent, we gave participants a pre-study questionnaire regarding their programming experience, including an assessment of their prior understanding of `final`. Participants in the `final` condition were asked to read three pages of documentation on `final`; participants in the Glacier condition completed a two-page paper-based tutorial on using Glacier. Participants were permitted to ask questions during this phase of the study. The remainder of the study consisted of four programming tasks in three different Java packages. Participants used the IntelliJ IDEA Community 2016.2 Integrated Development Environment (IDE) with Java 1.8 on a 15" MacBook Pro; we recorded audio and a video of the screen for analysis. We helped participants as needed with issues related to the computer system and IDE they were using, such as how to find a web browser and how to copy/paste, but did not answer questions about Glacier or `final`.

A study replication package is available [47], including all materials that were used in the study.

Task 1: making Person immutable. The Person package only included two classes: Person and Address. We asked participants: "Please make any necessary changes so that 'Person' in the 'person' package is immutable. After you're done, there should be no way to change an instance of a class after it is created." Participants had 22 minutes to complete this task.

```
public class Person {
  String name;
  Address address;
  ...
}
```

We expected that some participants in the `final` condition would neglect to mark Address as `final`.

Task 2: making Accounts immutable. The Accounts package represents all of the user accounts on a computer system. We asked participants: "Please make any necessary changes so that 'Accounts' in the 'useraccounts' package is immutable. After you're done, there should be no way to change an instance of a class after it is created." Participants had 20 minutes to complete this task.

```
public class Accounts {
  User [] users;
  ...
}
```

We expected that some participants in the `final` condition would neglect to modify the `User` class; in addition, making this class immutable required defensively copying the `users` array because there is no way in Java to make array elements `final`, and we expected that some participants would forget.

Glacier participants who did not complete tasks 1 and 2 in the allotted time were told how to finish because otherwise the resulting compiler errors would interfere with the next tasks.

Revision with advice. After they completed tasks 1 and 2, participants in the `final` condition were given a copy of page 73 from *Effective Java* [23], which outlines how to make a class immutable:

1. Don't provide any methods that modify the object's state.
2. Ensure that the class can't be extended.
3. Make all fields `final`.
4. Make all fields `private`.
5. Ensure exclusive access to any mutable components.

Participants could ask any questions for clarification; then, they were told they could revise their work from the previous tasks.

Task 3: FileRequest.execute(). We were interested in whether using Glacier would prevent programmers from creating security vulnerabilities in their software. The Java getSigners() bug [124] involved a private array being returned from an accessor, enabling any client to modify the contents of the array. We replicated the structure of the getSigners() bug in the context of the code from the previous task. Participants were told: "A FileRequest represents a request for a particular file from a web server, represented by a WebServer object. Normally, third-party clients implement their own types of requests, so it is important that the Accounts object that a Request gets access to is secure. As a test of the Accounts system, please implement FileRequest.execute() so that it does the appropriate access checks before granting access. In the process, you will need to implement User.getAuthorizedFiles()." Participants had 20 minutes to complete this task.

Although implementing User.getAuthorizedFiles() was stated as an incidental task, we were primarily interested in whether participants who used `final` remembered to copy the private array, `authorizedFiles`. Neglecting to do so would result in a security vulnerability similar to the `getSigners()` bug, since then any client of User could change which files a User was authorized to access. In the Glacier condition, participants could either copy the private array before returning it, or change the return type to return an `@Immutable` array; by enforcing transitive immutability of User, Glacier would identify all unsafe handling of the array. Participants in the `final` condition who did not copy the array but told the experimenter they were done with the task were given a sample of exploit code and then given an opportunity to revise their solution.

Task 4: HashBucket.put(). We wanted to know whether Glacier could prevent users from accidentally inserting mutation into existing immutable classes in real-world-like situations; is this an error that many programmers make without Glacier? We based our task on bug #1297 [161] in BaseX, which is an open-source XML database [160]. In that bug, the `delete` method on an implementation of an immutable hash map incorrectly modified the old hash map's data structures. In order to replicate this in a user study, we simplified the implementation to use a much simpler data structure while leaving the external API and comments in place as much as possible. The result was code that should be substantially easier to read and understand than the original and included many hints that the class was immutable, such as the fact that all modification methods returned a new object and the fact that the implementations of the provided methods made extensive use of copying.

	final	Glacier
Correctly enforced immutability in Person	0/10[1]	10/10
Correctly enforced immutability in Accounts	0/10[1]	9/10[2]
FileRequest.execute() tasks without security vulnerabilities	4/8	7/7
HashBucket.put() tasks without bugs	3/10	7/7

Table 5.1: Summary of user study results

We gave participants our hash map, which was implemented with an array of buckets, each of which contained lists of keys and values. The instructions to participants included: "HashBucket.put() is only partially implemented. Please finish the implementation by replacing the placeholder 'return this' with the right code." Since this task was last, we gave participants as much time as they needed to complete this task except when the total study period was exhausted. Users in the `final` condition who erroneously modified the old object's data structures and declared that they were done, if time allowed, were given an additional test case that exhibited the problem and the opportunity to fix their implementations.

5.4.2 Participants

We solicited participation from several different degree program mailing lists at Carnegie Mellon and from our acquaintances. Most of the participants were either Master's or Ph.D. students. We recruited 20 Java programmers (six of whom self-identified as female and 14 as male), and paid them $15 after they completed our study, which took about an hour and a half. Their programming experience ranged from four to nineteen years, with a mean of 9.5 years. Everyone had at least a year of Java experience; the mean was three years. They had an average of two years of professional experience writing software. We also asked participants to self-report their level of Java expertise, selecting from "novice," "intermediate," and "expert." Three participants identified themselves as experts; the rest considered themselves intermediate-level. Fifteen (75%) had used Java annotations before; eighteen (90%) had used `final` before.

We asked participants five questions about the behavior of `final`; the average score was 3.45/5. 11 participants knew what `final` meant when specified on a class, and 11 knew what `final` meant when specified on a method declaration. 17 participants knew that `final` does not forbid assignment in a constructor; 17 knew that it forbids assignment in setters; 13 knew that it does not forbid calling setters on final fields. We found no relationship between experience using Java and the number of these questions participants answered correctly.

5.4.3 Results

Results are summarized in Table 5.1. The denominators vary because some participants did not complete all tasks.

[1] After participants made corrections.

[2] Including extra time and compensating for searching; see § 5.4.3.

Error	# users
Provided mutating methods	0
`Person` not `final`	6
`Address` not `final`	10
`Accounts` not `final`	2
`User` not `final`	9
Fields of `Person` not `final`	2
Fields of `Address` not `final`	6
`Accounts.users` not `final`	1
Fields of `User` not `final`	4
Fields of `Person` not `private`	4
Fields of `Address` not `private`	8
`Accounts.users` not `private`	2
Fields of `User` not `private`	7
Omitted copying `users` in `Accounts` constructor	4
Omitted copying `users` in `Accounts.getUsers()`	2
Omitted copying `authorizedFiles` in `User` constructor	8

Table 5.2: Errors made by participants using `final` for immutability that remained after revision. Errors consist of failures to follow the advice in *Effective Java* [23].

After participants revised their code according to the advice from *Effective Java*, we counted errors (shown in Table 5.2) participants made with respect to that advice. Despite having the recommendations available while editing, every participant using `final` made mistakes. Two users made no non-transitive mistakes (i.e., mistakes directly in the `Person` and `Accounts` classes). No users remembered to make `Address` final, even though an instance of `Address` was used in `Person`. We conclude that enforcing immutability with Java as it currently exists is too complicated and error-prone for Java programmers to do effectively.

We stopped one user in each task at the time limits (22 minutes and 20 minutes, respectively). The average initial (pre-revision) time for Person and Accounts were 4 and 6 minutes, respectively. Participants spent an average of 6 minutes revising after receiving the *Effective Java* page. Among participants who said they were done with both tasks, the total average time, including revisions, was 15 minutes.

Making Person and Accounts immutable with Glacier. All of the Glacier participants successfully annotated Person with Glacier. Three did not finish modifying Accounts within 20 minutes, though one was given additional time and succeeded after 6 extra minutes. A common obstacle in the Accounts task was initializing an immutable array. The starter code included:

```
String[] files = {"RootUserBankAccounts.txt"};
```

Unfortunately, Java forbids annotations in an obvious place:

```
String @Immutable [] files =
  @Immutable {"RootUserBankAccounts.txt"}; // ERROR
```

Participants needed to write instead:

```
String @Immutable [] files =
  new String @Immutable [] {"RootUserBankAccounts.txt"};
```

Most users solved this with an Internet search, but the time required to do this was very variable. If we ignored this time, then two additional participants (9/10) would have succeeded within 20 minutes.

Several of the earlier participants did not annotate one of the classes until the next task due to a problem with the build system, in which it failed to rebuild all files that depended on the changed files; we later revised the instructions to avoid this problem. Correcting for this and deducting the time spent on array initialization resulted in an average total annotation time (across both tasks) of 11 minutes; applying these corrections to `final` users results in an average of 14 minutes for those users. The difference compared to the average time for `final` users is significant with $p < 0.1$ (Wilcoxon rank sum test).

FileRequest.execute(). Seven of the Glacier users successfully completed this task. One participant encountered an unrelated build system bug; two did not finish within 20 minutes. One of these two misinterpreted the starter code and attempted to implement `getAuthorizedFiles()` as a much more complex method than the accessor that was intended. Eight of the `final` users said within 20 minutes that they were done. Though Glacier prevented any security problems for the Glacier users, four of the `final` users (half of those who finished) did not copy the private authorizedFiles array, causing a security vulnerability similar to the Java `getSigners()` bug.

HashBucket.put(). All of the `final` users said they completed the task within 27 minutes, but they required up to an additional 11 minutes to fix their bugs after we showed them the additional test case. The average total time for `final` users was 18 minutes; the average total time for Glacier users who finished was 14 minutes. The difference in times was not significant. Seven of the `final` users incorrectly modified the HashBucket's internal data structures, resulting in a bug. In addition, six `final` participants returned the existing HashBucket instance rather than creating a new one. One Glacier user gave up after 29 minutes, having gotten an error from Glacier after trying to modify an immutable array and could not figure out another way to solve the problem. In addition, two Glacier users had already exhausted the overall study period and could not be given enough time to finish. Overall, 3 of 10 `final` users completed the task correctly; Glacier detected the problem statically, and 7 of 9 Glacier users who started the task completed it successfully.

5.4.4 Discussion

We had hypothesized that participants would spend significantly less time specifying immutability with Glacier than with `final` because using Glacier required only adding annotations, whereas using `final` required making several kinds of changes, such as copying arrays in constructors. Variance in time was high in both tasks, as is typical in studies of programmers [90]. For example, some users (particularly in the `final` condition) wrote test code to see if they could cause data to be mutated; others wrote no tests (time spent writing and executing tests was included in the task times above). In addition, `final` users would have spent more time if they had completed all of the work required to do the tasks correctly. However, even if Glacier did not save users any time in specifying immutability relative to `final`, it likely took users the same amount of time to

enforce a much stronger property while avoiding mistakes.

One might have expected `final` participants to make fewer errors than they did, since all of the participants in that condition had used `final` before. However, some of the participants had never attempted to use `final` to enforce immutability. One participant remarked when starting to read the `final` documentation: "I've only used `final` on integers before, so this will be instructive." Some participants vocalized considering and rejecting Bloch's advice, for example reasoning that since a class had no setters, it was immutable, so other changes (such as making the class `final`) were unnecessary. When using `final` for immutability, then, it is not sufficient to say that a class should be immutable; one must say exactly what kinds of future changes the class should be robust to.

In the `final` condition, the requirement to defensively copy arrays in constructors and accessors was particularly problematic. For example, 80% of the participants omitted a defensive copy of the `authorizedFiles` array in the `User` constructor. Some participants complained about the performance impact of this strategy: after implementing defensive copying in `getUsers()`, one participant remarked, "I'm not really happy. If there's a lot of users and `getUsers` is called frequently... that will slow down the performance." Two participants, after reading the *Effective Java* page, asked for an explanation of why defensive copies were necessary, but even after hearing the explanation, one of these two users omitted required defensive copying. We conclude that although following the advice would result in certain protections against mutation, few users can successfully apply the advice to even a simple programming project, even when given the advice immediately before needing to use it, and even with access to the recommendations while programming; we believe the problem is one of complexity, lack of enforcement, and lack of understanding that the recommendations are relevant.

Some Glacier users reported that the two annotations on arrays—one on the array itself and one on the component type—were confusing. Though this a fundamental aspect of containers, we believe that one of the reasons users faced difficulty is that the design of arrays (a relatively old feature) is inconsistent with the approach used in generic classes, in which the component type is specified in angle brackets rather than next to the container name. The user study did not include tasks involving annotated type parameters; our experience suggests that *using* them is straightforward, but writing parametrized classes can be difficult due to the need to specify type parameter bounds.

Our impression from observing participants was that Glacier's error messages tended to force users to understand transitivity; `final` users who misunderstood our definition of immutability received no such feedback. Future studies should distinguish between behavioral changes caused by a clearer understanding of immutability (fostered by training or tooling) and behavioral changes caused by compiler enforcement.

5.4.5 Limitations

The main threats to validity are of our study are due to the simplicity and limited nature of our tasks and the relative inexperience of our participants. Likewise, our participants came from a relatively narrow set of backgrounds. It is possible that more-expert Java programmers would have been able to use `final` more successfully and that the training we provided for `final` was insufficient. We think it is unlikely that programmers would be *less* likely to incorrectly

mutate immutable structures in a more complex codebase than the one we provided, but perhaps more-expert programmers would be better at identifying the implicit immutability requirements. We selected our tasks to expose opportunities to mutate structures incorrectly; though we have shown that these tasks do result in incorrect mutation, the fraction of real-world programming tasks that are similar is unknown.

5.5 Conclusion

We have designed, formalized, and implemented Glacier, which enforces transitive class immutability in a Java annotation processor. We conducted a user study and found that Java programmers could generally specify immutability effectively with it; in contrast, Java programmers in our study could not use `final` to specify immutability even though they had advice on how to do so. We also found that programmers incorrectly mutate immutable data structures when they only have `final`, whereas Glacier detects those errors statically. Glacier represents a promising approach to enforcing immutability in real-world Java software and implements a model that could be extended to other languages as well.

Glacier's development process serves to show the value of applying user-centered methods to the design of programming language extensions. By basing the language design on formative studies, we were able to focus the design on features that would be most useful; by evaluating the language with users, we were able to show both that users could use the tool effectively and that it addressed a problem that would likely arise frequently in real-world-style code.

In the next chapters, we will show how we extended the user-centered design approach to a novel programming language, Obsidian, which is targeted at blockchain programming.

Chapter 6

Introduction to Obsidian[1]

Developing Glacier (chapter 5) provided an opportunity for exploring using several HCI techniques for designing and evaluating programming languages, such as *interviews* and *summative usability studies*. However, Glacier was an extension to an existing programming language. For the next project, we were interested in exploring a larger programming language design problem in a more novel space in order to have more flexibility to introduce novel language design approaches.

In *blockchains* [93], a decentralized network of computers maintains system state and executes transactions. Blockchains support deploying *smart contracts*, which are programs that can maintain state. Blockchains have been proposed for high-stakes applications such as financial transactions, health care [91], supply chain management [99], and others [67], and are an ideal testbed for a new language design process. The need for a safer programming language is motivated by the history of security vulnerabilities, through which over $80 million worth of virtual currency has been stolen [81, 185]. However, ordinary programmers and software engineers need to be able to write blockchain applications; it does not suffice to assume that the developers will be experts in formal verification or that companies will invest the resources required to formally verify that their programs are correct. Instead, we sought a more *lightweight* approach that provides additional safety guarantees at low cost to developers.

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*.

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* [194] to statically ensure that objects are manipulated correctly according to their current states, and uses *linear types* [214] 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.

This chapter contributes a *requirements analysis* for smart contract languages. It also summa-

[1] Portions of this chapter previously appeared in [48, 51].

rizes the *final* design of Obsidian using an example. Chapter 8 describes the methods we used to iterate on our initial design to arrive at the final design. Chapter 9 describes the language formally. Chapter 10 describes case study evaluations of Obsidian, and chapter 11 describes the empirical user evaluation.

6.1 Requirements for Smart Contract Languages

We began work on Obsidian by eliciting requirements for successful smart contract languages by observing the application domain and the history of the community's experience with existing languages.

6.1.1 Strong static safety

Once a smart contract is deployed and users rely on the deployed contract, bugs in the deployed contract can be impossible to fix, since the nature of some blockchains is such that the contract cannot be modified after deployment. Furthermore, smart contracts may implement high-stakes software governing valuable resources, such as cryptocurrencies, or maintaining important business records. Bugs in smart contracts undermine the very purpose of the blockchain, which is to facilitate transactions among parties that have not established trust.

Delmolino et al. found that novices typically make several serious errors when writing smart contracts [59]; although the study was done with Serpent, which predated Solidity, popular blockchain languages have not been designed to prevent the bugs that Delmolino et al. found, such as errors in encoding state machines resulting in loss of money.

As a result, we argue for strong, compile-time mechanisms to eliminate as many classes of bugs as is practical. Work on static smart contract checking, such as Oyente [121] and MadMax [83], aims to find instances of particular classes of bugs. But these kinds of static analysis may not have the same kinds of impact on programmer's reasoning as deeper language design decisions. Indeed, correctness by construction may be a more direct means of attaining safety, since it demands that programmers structure their artifacts in a way that avoids classes of bugs. It is better, then, to combine careful language design, which shapes how people think, with static analyses that can detect other bugs that the language cannot prevent. For example, commonly-used languages offer type systems that do not facilitate reasoning about assets, which can be owned and consumed. This leads to bugs in which assets are accidentally either consumed more than once or lost forever, as has been shown to occur [59]. Likewise, traditional languages do not facilitate compile-time reasoning about the states that objects are in, even though smart contracts typically implement state machines that support different transactions depending on the state. This can lead to bugs in which transactions are invoked on contracts that are in inappropriate states. Prior work showed that programmers benefit from typestate annotations when using object protocols [196]; since these protocols are common in blockchain systems, we wanted to see whether typestate might benefit programmers when writing implementations as well.

6.1.2 User-centered design

Blockchain software development is fundamentally a *human* process, involving human developers who must create applications to meet *human* needs [43]. A smart contract language, then, is an interface by which people can specify behavior, and is subject to the principles of human-computer interaction.

Many programming language designs and tools for software engineers aim to reduce the difficulty of writing programs, but unfortunately, most of the tools in use today were not formally evaluated with users. Instead, designers guessed what would be beneficial for users and provided it. But the users of the tools may have different needs than the designers, so it does not suffice for designers to create tools for themselves and hope that others will learn to use the tools effectively. We argue that software engineers, being people too, are amenable to the methods of human-computer interaction [131] and that designers of programming languages should use a wide variety of techniques in order to make their tools more usable [43].

Many approaches developed in the programming language research community are not adopted by developers because they are too hard to use or are impractical for real-world usage. For example, Bhargavan et al. [17] proposed formal verification of smart contracts. Though many researchers are working on this, formal verification is still impractical for most programmers to use.

As a medium for our research on user-centered design of programming languages, we also sought to use Obsidian to advance the science of programming language design. As a result, we made high-level design choices that would enable direct comparison with existing approaches, such as Solidity. For example, making Obsidian an object-oriented language with a similar syntactic structure as Solidity meant that we could recruit from the same population in user studies comparing both languages.

6.1.3 Blockchain-agnosticism

As of this writing, there were at least 28 different blockchain platforms [222]. An author of a smart contract may need to deploy on one blockchain platform initially, and then migrate as the blockchain market matures. Even Ethereum, which is the longest-running and most popular smart contract platform, plans significant changes in Ethereum 2.0, which may cause some users to migrate to or from Ethereum. Each platform supports a particular set of languages; currently-supported languages include Java, Go, Kotlin, Solidity, C++, WebAssembly, Python, JavaScript, Liquidity, and Rust (among others). A new smart contract language that only works on one platform locks itself to an uncertain future. More importantly, programs last a long time, and implementations need to be robust to changes in underlying infrastructure.

Instead of committing to a particular platform, designers should make a small number of safe assumptions about the nature of blockchain platforms, allowing developers to create stable software against a simple abstraction. By doing so, language designers might improve safety and usability relative to general-purpose languages. Languages might rely on the following properties that most blockchain systems have in common:

Sequential execution: Blockchain platforms typically support neither parallelism nor concurrency, simplifying reasoning relative to systems that do support either feature.

Deterministic evaluation: Because all peers must obtain the same result when executing transactions, smart contracts cannot depend on arbitrary external API invocations.

High cost of computation: Computation on the blockchain is substantially more expensive than off-chain computation, so although transactions are typically small, they need to be exceptionally cheap. Public blockchains require users to pay cryptocurrency according to the computational cost of their transactions, providing additional motivation to keep transactions cheap and also motivating a need to predict transaction costs in advance of execution.

Unpredictable transaction ordering: Clients submit transaction requests to blockchains without knowing what transactions will occur between their submission and the transactions' executions. If an intervening transaction violates an implicit precondition of a transaction, the results of the transaction might be surprising to the user who issued it. [121]

Cryptography needed to maintain secrecy: Blockchains permit a collection of nodes to see all the data on the blockchain. Any data that should remain secret from some of these nodes must be encrypted. This requires cryptographic techniques, such as cryptographic commitments [10]. This difficulty comes up frequently; for example, a gambling application might need to allow players to place secret bets, but if the bets are stored naïvely, the bets will be public.

Off-chain interaction: Client software that executes off-blockchain needs to interact with deployed smart contracts. Some existing approaches, such as that in Ethereum, limit the APIs to those that only take primitive arguments, stifling the expressiveness of the APIs. New languages should facilitate well-designed, expressive APIs by allowing arbitrary data structures to be passed and returned. New languages should make it easy to develop client software that interacts with blockchains.

Storage: Some blockchain platforms, such as Hyperledger Fabric, require users to manually save and restore data from the ledger as key/value pairs. This allows fine-grained control but also requires programmers to expend significant effort and offers the potential for bugs. Languages should facilitate automatic, efficient storage of smart contract state.

6.2 Obsidian Language Summary

Detecting bugs was our initial objective, so we considered bugs, such as the DAO hack [55], which resulted from a reentrant invocation in which a contract allowed itself to be invoked while in an inconsistent state. We also analyzed characteristics of proposed blockchain applications. In general, we observed that proposed blockchain applications typically maintain high-level state, which governs which operations are safe.

Objects in smart contracts frequently maintain high-level state information [71], with the set of permitted transactions depending on the current state. For example, an `Auction` might be `Open` or `Closed`, and the `bid` transaction can only be invoked on an `Open` auction. Prior work showed that including state information in documentation helped users understand how to use object protocols [196], 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.

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, in order to improve safety relative to traditional designs, Obsidian omits inheritance, which is error-prone due to the *fragile base class problem* [127]. We leveraged some features of the C-family syntax, such as blocks delimited with curly braces, dots for separating references from members, etc., 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 [214], 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; for another example, see [201]). A particular innovation in this approach is the fusion of linear references to assets with affine references to non-assets. Thus, references to non-assets can be discarded as needed, but owning references to assets cannot be discarded without using `disown`. Whether a reference is linear or affine depends on the declaration of the type to which the reference refers. States and contracts 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 (or any 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.

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, it may have `Shared` and `Unowned` aliases. Examples of some of these scenarios are shown in Figure 6.1 to provide some intuition. A summary of modes is shown in Table 6.1.

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`

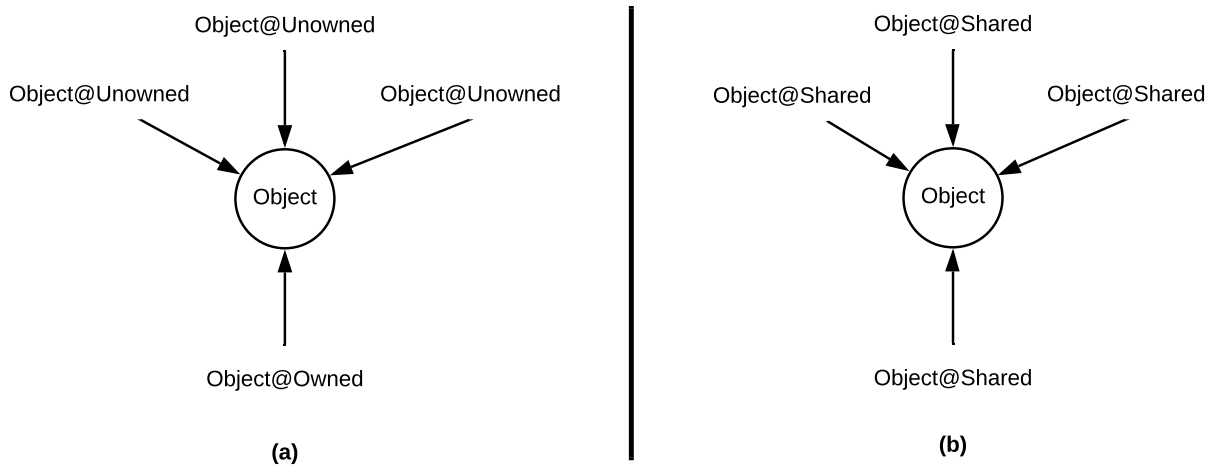

Figure 6.1: Some common aliasing scenarios. (a) shows an object with one owner; (b) shows a shared object.

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`.

In order to leverage those observations, we became interested in a *typestate*-oriented approach [5], in which *states* of objects are incorporated into types. For example, rather than merely having a `LightSwitch` type, we can have `LightSwitch@On` be the type of a reference to an object that is in the `On` state. Then, if the user attempts an invalid operation, such as turning on a switch that is already on, the compiler can issue an error.

Featherweight Typestate [78] 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 chapter 8. The new language requirements we identified motivated the design of a new formalism; we designed *Silica* (§ 9.4), 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.

Typestate-based types are in a class called *linear types*. Unlike traditional types, linear types can change as operations are performed. For example, invoking `turnOff()` on a reference of type `LightSwitch@On` *changes the type* of the reference to `LightSwitch@Off`. Conveniently, linear types are also what are needed to ensure that assets are never lost. Obsidian includes *owned* objects: for each owned object, there is an object that owns it via an owning reference. If a

local variable that owns an asset goes out of scope, the compiler emits an error message. Fields that own assets can only exist in contracts that are themselves assets. This way, each asset always has an owner.

Smart contracts commonly manipulate *assets*, such as virtual currencies. Some common smart contract bugs pertain to accidental loss of assets [59]. 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, 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.

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

Table 6.1: A summary of modes in Obsidian.

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 $\gg$ symbol separates the initial mode from the final one. In the example of Figure 6.2, the signature of `buy` (lines 27-28) 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 11), 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.

A formal description of Obsidian is in chapter 9. However, Figure 6.2 shows some of the key features of Obsidian using the example of a *tiny vending machine* (TVM). `TVM` is a `main` contract, so it can be deployed independently to a blockchain. A `TVM` 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 }
```

Figure 6.2: A tiny vending machine that shows key features of Obsidian.

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.

As another example of the relevance of linearity in smart contracts, we wrote an Obsidian implementation of the ERC-20 token standard [213], 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, 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* [29] 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.

```
import "Dict.obs"
import "Integer.obs"

asset interface ObsidianToken {
  transaction getValue() returns int;
  transaction merge(ObsidianToken@Owned >> Unowned other);
  transaction split(int val) returns ObsidianToken@Owned;
}

asset contract ExampleToken implements ObsidianToken {
  int value;

  ExampleToken@Owned(int v) {
    value = v;
  }

  transaction getValue(ExampleToken@Unowned this) returns int {
    return value;
  }

  transaction merge(ObsidianToken@Owned >> Unowned other) {
    value = value + other.getValue();
    disown other;
  }

  transaction split(ExampleToken@Owned this, int val) returns ExampleToken@Owned {
    if (val > value) {
      revert ("Can't split off more than the existing value");
    }
    ExampleToken other = new ExampleToken(val);
    value = value - val;
    return other;
  }
}

// ERC20 has been slightly adapted for Obsidian, since Obsidian does not have
```

```
//   a built-in authentication mechanism.
asset interface ERC20 {
  transaction totalSupply() returns int;
  transaction balanceOf(int ownerAddress) returns int;
  transaction transfer(int fromAddress, int toAddress, int value) returns bool;

  // - allow ownerAddress to withdraw from your account,
  //   multiple times, up to the value amount.
  transaction approve(int ownerAddress, int fromAddress, int value) returns bool;

  // Returns the amount of allowance still available.
  transaction allowance(int ownerAddress, int fromAddress) returns int;

  // Transfers tokens from an allowance that has already been granted.
  transaction transferFrom(int senderAddr, int fromAddr, int toAddr, int value)
        returns bool;
}

main asset contract ExampleTokenBank implements ERC20 {
  int totalSupply;
  Dict[Integer, ExampleToken]@Owned balances;

  // map from fromAddress to (map from spender to amount)
  Dict[Integer, Dict[Integer, Integer]@Owned]@Owned allowed;

  ExampleTokenBank@Owned() {
    totalSupply = 0;
    balances = new Dict[Integer, ExampleToken@Owned](new IntegerComparator());
    allowed = new Dict[Integer, Dict[Integer, Integer]@Owned](new IntegerComparator());
  }

  transaction totalSupply() returns int {
    return totalSupply;
  }

  transaction balanceOf(int ownerAddress) returns int {
    Option[ExampleToken@Unowned] balance = balances.peek(new Integer(ownerAddress));
    if (balance in None) {
      return 0;
    }
    else {
      return balance.unpack().getValue();
    }
  }

  transaction transfer(int fromAddress, int toAddress, int value) returns bool {
    Integer fromIntegerAddress = new Integer(fromAddress);
    Option[ExampleToken@Owned] fromBalance = balances.remove(fromIntegerAddress);
    if (fromBalance in None) {
      return false;
    }
    else {
      ExampleToken fromTokens = fromBalance.unpack();
      if (value <= fromTokens.getValue()) {
        Integer toIntegerAddress = new Integer(toAddress);
        Option[ExampleToken@Owned] toBalance = balances.remove(toIntegerAddress);
        ExampleToken toTokens;
        if (toBalance in Some) {
          toTokens = toBalance.unpack();
        }
        else {
          toTokens = new ExampleToken(0); // 0 value
        }

        ExampleToken tokensToMove = fromTokens.split(value);
        toTokens.merge(tokensToMove);
```

```
      balances.insert(toIntegerAddress, toTokens);
      balances.insert(fromIntegerAddress, fromTokens);

      return true;
    }
    else {
      // Insufficient funds available.
      balances.insert(fromIntegerAddress, fromTokens);
      return false;
    }
  }
}

// Records a new allowance. Replaces any previous allowance.
transaction approve(int ownerAddress, int fromAddress, int value) returns bool {
  Integer ownerAddressInteger = new Integer(ownerAddress);
  Option[Dict[Integer, Integer]@Owned] ownerAllowancesOption =
      allowed.remove(ownerAddressInteger);

  Dict[Integer, Integer] ownerAllowances;
  if (ownerAllowancesOption in None) {
    ownerAllowances = new Dict[Integer, Integer@Owned](new IntegerComparator());
  }
  else {
    ownerAllowances = ownerAllowancesOption.unpack();
  }

  Option[Integer@Owned] oldAllowance = ownerAllowances.replace(
                                        new Integer(fromAddress),
                                        new Integer(value));
  allowed.insert(ownerAddressInteger, ownerAllowances);

  // Options are assets because they CAN hold assets,
  // but this one doesn't happen to do so.
  disown oldAllowance;
  return true;
}

transaction allowance(int ownerAddress, int fromAddress) returns int {
  Option[Dict[Integer, Integer]@Unowned] ownerAllowancesOption =
      allowed.peek(new Integer(ownerAddress));
  switch (ownerAllowancesOption) {
    case None {
      return 0;
    }
    case Some {
      Dict[Integer, Integer@Owned] ownerAllowances = ownerAllowancesOption.unpack();
      Option[Integer@Unowned] spenderAllowance =
          ownerAllowances.peek(new Integer(fromAddress));
      if (spenderAllowance in None) {
        return 0;
      }
      else {
        return spenderAllowance.unpack().getValue();
      }
    }
  }
}

// senderAddress wants to transfer value tokens from fromAddress to toAddress.
// This requires that an allowance have been set up in advance and that
// fromAddress has enough tokens.
transaction transferFrom(int senderAddr, int fromAddr, int toAddr, int value)
            returns bool
{
  int allowance = allowance(senderAddress, fromAddress);
```

```
    if (allowance >= value) {
      int newAllowance = allowance - value;
      bool transferSucceeded = transfer(fromAddress, toAddress, value);
      if (!transferSucceeded) {
        // Perhaps not enough tokens were available to transfer.
        return false;
      }
      approve(senderAddress, fromAddress, newAllowance);

      return true;
    }
    else {
      return false;
    }
  }
}
```

Chapter 7

Permission Taxonomy and Permission System Design

7.1 Permission Taxonomy

In designing Obsidian, we conducted an analysis of the space of permission systems so that we might understand the context in which we were designing Obsidian. This analysis helped us make preliminary design decisions and identify questions that we could address in user studies.

After Boyland [28], a *permission* confers the ability to perform certain operations on or with the referenced object, but that ability may be consumed when it is used. Our taxonomy of permission systems is organized around *user-visible concepts* and focuses on different *ideas* that the system allows the user to express. The taxonomy considers both static and dynamic enforcement. There is also a question of what each permission pertains to: should a permission apply to *references* or *objects*?

Permit What does the permission allow the holder to do? Possibilities include equality testing the reference, equality testing the object, reading object fields, mutating the object, mutating the reference (for example, reassigning it to refer to a different object), and calling methods.

Deny What does the permission deny, either to the holder or to others? Options are the same as for Permit. For example, in Obsidian, a static typestate specification on a reference implies that other aliases to that same object cannot mutate its state.

Apply To what does the permission apply: types (in which case it applies to every instance of those types), individual objects or values, or references?

Require What requirements are imposed on the holder of the reference? For example, if the reference represents ownership, which is linear, then holding the reference imposes a requirement to transfer the reference somewhere else before it goes out of scope.

Know What knowledge does the permission imply? For example, it might specify typestate statically or dynamically, or it might assure that the referenced object will not be modified (immutability).

Extent What is the extent of a permission? What portion of the referenced object graph does the permission pertain to: the entire transitively-referenced state, just one object, or even just a

specific field?

Compose How can different permissions be composed? For example, fractional permissions can be recombined into a full permission [28]. Can permissions to different components of an object be combined?

Test How can a dynamic operation be used to establish more knowledge than is available statically? For example, a dynamic state check could allow safe invocation of state-specific operations even when the state cannot be determined statically.

Parameterize How can programmers reason generically about permissions, e.g., with parametric polymorphism? Is it possible to have parameters that represent permissions?

Use What operations are permitted on permissions themselves? Can permissions be dropped, duplicated, or exchanged for other permissions?

Transfer How is a permission on one reference to an object transferred to another reference? When a permissioned reference is passed as a parameter, which permissions go with the reference? Does it depend on the reference, formal parameter, or call? Is there a way to pass the permission temporarily so the caller's permission is restored when the call returns?

Our goal was to develop a system that is sufficiently expressive for real-world scenarios but also simple enough that people can use it effectively. Though user studies could help evaluate the usability of a particular design, it would be impractical to do enough user studies to cover all possible designs. Instead, for each dimension, we show below the relevant design *decisions* as well as *questions* for user study. We prioritized the most important questions to evaluate rather than attempting to answer every question with a user study. We also describe the outcomes for each question, which we arrived at after the design work in § 7.2 and after the various user studies (chapter 8).

Permit Consistent with existing OO languages, `Shared` values can be mutated via any reference. Likewise, references whose types specify typestate permit invocation of appropriate transactions.

Deny If a reference to an object specifies typestate, all other aliases must deny state mutation (otherwise the system would be unsound). Transaction invocations do not type check if any of the transaction arguments are not known to be in appropriate states, given the transaction signature.

Apply Should a permission apply to contracts (so that every instance of the contract has the same permission), to objects, or to instances? Permissions apply to references, since user studies showed that explicit permission annotations were helpful (§ 8.2.3). The `asset` declaration on a contract affects every instance of that contract, since our study of the application space suggests that contracts typically have either all-linear instances or all-affine instances.

Require The holder of an owning reference to an asset must either transfer ownership before the reference goes out of scope or explicitly discard the reference. This requirement is implied by the definition of linearity.

Know Typestate-bearing references convey static knowledge that the referenced object is in a given state; this approach gives the strongest safety properties, facilitating conservative transaction invocation (transactions can only be invoked when they are statically known to

be available). How should these be written? Should there be a permission annotation, or does the typestate specification suffice? For concision and to reduce cognitive burden, we decided that typestate-specifying references would not need to also specify a permission.

Extent Specifications and restrictions pertain only to the referenced objects because they specify operations that are permitted or denied on those objects, consistent with the usual semantics for type systems. Should non-owning aliases prohibit all mutation, or just prohibit changing typestate? For soundness, only prohibiting changes to typestate is required, but perhaps it is simpler and thus more usable to prohibit all mutation. We did not prioritize this question for explicit investigation in a user study, but our participants did not experience any apparent problems with this design.

Compose Because objects have independent state machines and are owned independently, permissions for different objects are orthogonal.

Test Static knowledge is insufficient in some cases because necessary guarantees cannot always be obtained statically [115]. Therefore, Obsidian needs to support dynamic state checks. What should the syntax be? Options include `switch`-like (a structured approach providing warnings when cases are omitted) or `if`-like (providing a Boolean expression that tests whether a contract is in a particular state). We added both approaches in the concrete Obsidian language, but did not prioritize evaluating this particular choice.

Parameterize Adding parametric permissions would increase language complexity and potentially decrease usability, but generic programming might be more concise. We added contract and permission parameters to enable development of reusable containers, such as linked lists, since that would otherwise be impossible.

Use Linear assets sometimes need to be discarded intentionally. For example, a holder of a subway token object might choose to never use it. Thus, Obsidian allows explicit discarding of owning references. But what should this operation be called, and what should the syntax look like? We added `disown` to address this need.

Transfer Our initial design implicitly transferred ownership on invocation when the formal parameter was of owning type. Would this be confusing? If so, would an explicit approach be better? Indeed, we found in user studies that this approach was confusing (§ 8.2.3); we changed to an explicit syntax in formal parameters for representing changes in permissions and added a checked permission assertion.

7.2 Decomposition of permissions for Obsidian

Given our usability objectives, we sought the simplest permission system consistent with the above decisions. First, we decomposed the space of permissions into primitive permissions that could be combined to express the necessary concepts. This facilitated a principled analysis of the space of permissions. Then, we enumerated all combinations to assess which of the primitive permissions should be exposed individually, which combinations should be given easy-to-understand names, and which combinations were nonsensical or not useful. We observed a trade-off between providing a small number of cohesive concepts and providing a maximally

expressive system.

The decomposition depends on whether permissions are properties of contracts, objects, or references. If they are properties of contracts, this might lead to a simpler language because there would be fewer combinations available. On the other hand, it may be too restrictive. For example, is it really the case that every contract type will be always shared or always owned? Furthermore, is it even the case that every *object* will be either shared for its whole lifetime? It seemed that some useful cases would involve tracking typestate for some period of time, during which ownership would be required, and then later making the object shared. To enable this use case, as well as to avoid forcing developers to create shared and owned versions of contracts that needed to be used in both ways in different situations, we selected the more expressive approach in which sharing is a property of references. Also, based on our understanding of how objects are likely to be used in smart contracts, we also decided that there would be asset and non-asset contracts, and analyzed the interactions between permissions, assets, and non-assets.

Enforcing linearity for assets entails restricting construction and destruction. Existing approaches (such as runtime checks in constructors or requiring capabilities to be passed to enable constructor execution) can restrict construction, so we focus here on mechanisms to prevent destruction of objects.

Table 7.1 shows the resulting permission space. References are **owning** or **unowning**; **durable** or **non-durable**; and **stateful**, **state-readonly**, or **state-unrestricted**.

These combine in interesting ways, especially since each kind of reference may refer to assets and non-assets. This separation is particularly interesting because it shows new kinds of restrictions that we might not have considered (and which would be interesting to explore in future work). In principle, one could have a durable reference to an object that is not an asset, enforcing part of linearity on arbitrary references. However, in Obsidian, to simplify the system, only a durable reference may own an asset. Thus, in Obsidian, durable references are exactly those that are owning references to contracts that were declared as assets.

Interpretations of the combinations are shown in Table 7.1. Table 7.2 shows the semantics of the combinations that are meaningful and useful. Analysis of these combinations resulted in the design for Obsidian shown in Table 7.3. Note that there is no keyword for *stateful*; instead, that is indicated by presence of state in the type specification. For example, `Bond@Sold` represents a reference to a `Bond` that is in the `Sold` state. Our initial language design supported borrowing explicitly, but the final design uses ownership transfer syntax (`Owned >> Owned`) to represent temporary ownership transfer instead.

An important property of Obsidian permissions is that if there is an owned reference to an object, there are no other owned or shared aliases. This is an important condition for soundness of the typestate specifications.

	Owned		Unowned	
	Durable	**Non-durable**	**Durable**	**Non-durable**
Stateful	Owned reference to an asset (1)	Owned, state-specifying reference (2)	Unsound; mutation through one of several aliases violates the others' guarantees	Unsound; mutation through one of several aliases violates the others' guarantees
State-readonly	Limited utility; an owned reference has no need to be restricted from modification	Limited utility; an owned reference has no need to be restricted from modification	Limited utility; why would one not want to lose a non-owned reference?	Counterpart of (owned, non-durable, stateful) (3)
State-unrestricted	Owned reference to an asset that has no states (4)	Simple exclusion (5)	A reference that cannot be discarded. Would imply an unowned asset, which is not useful.	Shared reference (traditional OOP semantics) (6)

Table 7.1: Permission combinations when sharing is a property of references. Gray backgrounds show unsound combinations. Numbers index into Table 7.2, which shows the properties of each numbered combination.

Permissions	**Assets**	**Non-assets**	**Holder of permission can**:			**Others can**:		
			change type-state	**drop ref.**	**specify type-state**	**change type-state**	**drop ref.**	**specify type-state**
(1) stateful asset	✓		✓		✓		✓	
(2) stateful		✓	✓	✓	✓		✓	
(3) state-readonly	✓	✓		✓		✓	✓	✓
(4) owned asset	✓		✓			✓	✓	
(5) owned		✓	✓	✓		✓	✓	
(6) shared		✓	✓	✓		✓	✓	

Table 7.2: Permission semantics of useful combinations. Numbers are references from Table 7.1.

Keyword	Specifies typestate	Asset		Non-asset	
		Permissions	**Meaning**	**Permissions**	**Meaning**
`Owned`		durable, owned	Linear reference	owned	Affine reference
`Shared`		N/A	N/A	shared	Standard semantics
`Unowned`		state-readonly	Aliases object that may be `Owned`	state-readonly	Aliases object that may be `Owned`
`Owned`	✓	durable, stateful, owned	Owned asset reference	stateful, owned	State-guaranteeing affine reference

Table 7.3: Obsidian permission system

Chapter 8

Formative Design in Obsidian[1]

8.1 Introduction

In this chapter, we describe techniques we used to triangulate data about individual language changes. Then, we describe the final design of Obsidian and explain the methods that we used to create and evaluate it.

We were interested in adapting traditional HCI methods to the context of the design of programming languages that target professional software engineers. Our high-level research question can be refined in terms of three research questions:

Naturalness: How can we obtain insights as to what language designs will be *natural* for programmers, given that we are trying to obtain particular static safety guarantees in the language?

Iteration: How can we iterate on a particular design to make it more effective for users?

Comparison: How can we compare two language designs to see which is more effective for users?

We wanted to apply known HCI methods, such as *natural programming* [132], *Wizard of Oz* [54], *interviews*, and *rapid prototyping*. However, we found that the study design process was very challenging due to the nature of programming and the complexity of the design space. These challenges included:

Training: How could we train participants in a new programming language in a short enough amount of time to make studies practical?

Recruiting: How could we recruit participants who have sufficient programming skill and whose results would generalize beyond the population of students, despite having limited access to professional software engineers?

High prototyping cost: How could we conduct user studies on programming languages that have only informal designs and no implementations, since cost of building working prototypes is high?

Variance and external validity: How would we mitigate high variance, which is typical in programming tasks, without constraining the tasks so much that they were no longer

[1] Portions of this chapter previously appeared in [45].

representative of real-world programming tasks?

The problems of variance and external validity were particularly relevant for quantitative studies, which needed to be practical in the context of our university setting. Programming tasks that are not extremely constrained tend to produce results with high variance, making statistical significance hard to obtain. On the other hand, tasks that are highly constrained suffer from low external validity, since real-world programming tasks are typically long and complex.

- By adapting the *natural programming* technique to allow *progressive prompting*, we were able to obtain both unbiased responses as well as data that were relevant to the particular designs we were considering.
- By *back-porting* language design questions to languages with which participants were familiar and by using the *Wizard of Oz* evaluation technique, we were able to obtain usability insights on incomplete designs, and *isolate* the design questions of interest from confounding variables.
- By dividing large tasks into multiple, smaller tasks, and by using pilot studies to set task time limits effectively in quantitative studies, we were able to reduce variance sufficiently to obtain meaningful results in complex programming tasks, which otherwise would have had very high variance.
- By recruiting participants who were *representative of at least some junior-level professional developers*, we were able to maximize external validity in our studies while still conducting them practically at a university setting. We were also able to show usability impacts of the language designs under consideration.
- By developing incremental tutorials with integrated practice opportunities, we were able to *teach* the languages to participants in a short period of time (for Obsidian, about 90 minutes was typical).

In this chapter, we describe how we integrated both formative and summative human-centered methods into the design process for Obsidian. We have two main contributions:

1. We show by example how we have adapted several formative study techniques, such as natural programming, Wizard of Oz, rapid prototyping, cognitive dimensions of notations analysis, and interview studies to inform the design of Glacier and Obsidian. We informed the results of these techniques with implications of the theory of programming languages to develop languages that were effective for users and also achieved our safety objectives. We found that our adapted methods were effective when used with particular kinds of study designs, which we described in chapter 12.
2. We show how we conducted summative usability studies on new programming languages. By developing ways of teaching the languages efficiently, effectively, and consistently, we were able to conduct usability studies of programmers using novel programming languages.

We summarize recommendations for others who want to use these methods in their own programming language design work. We believe that these recommendations may also be useful in other domains. Some properties of programming distinguish it from many kinds of tasks, but those same properties are shared with some other domains:

Problem-solving: Programming languages exist to facilitate problem-solving. However, problem-solving can be unpredictable [119]; in user studies, some participants typically complete

tasks almost instantly whereas others can spend hours working and still not finish. This large variance makes running quantitative user studies very challenging.

Range of working styles: Bergström and Blackwell described a diverse collection different approaches to programming problems [14], such as *bricolage/tinkering* and *engineering*. These different styles may be used even by different people using the *same language*, impeding a designer's attempts to anticipate a user's strategy or behavior.

High stakes: Errors when programming can contribute to serious real-world safety problems, e.g., in avionics or health care systems.

For example, CAD tools affect their users' creative processes [171]; likewise with process engineering tools [30] and even drug design tools [193]. All of these domains involving expert problem-solving by a variety of different people with high costs of failure.

8.2 Formative studies for Obsidian

In this section, we describe studies that helped us identify a suitable design and iterate on our initial design ideas for Obsidian. For each study, we identify our research questions, methodology, and results. We started by assuming that we would use *typestate* to achieve the desired safety guarantees but that expressing typestate in a usable way would require substantial iteration with users. The latter assumption was based on past work on typestate systems, such as Plural [19] and Plaid [198], which researchers had found were difficult for users to use. All of the studies were approved by our IRB. Because we needed skilled programmers, we recruited from appropriate academic programs, by posting flyers, and by contacting our acquaintances. Except where noted below, we paid participants $10/hour for participating. Materials used in the studies can be found in the supplement.

Although chapter 6 uses the final version of the language, because the formative studies were done earlier, they use code from earlier versions of the language. In this way, the reader can see how we changed the language as a result of the user studies. For example, Figure 8.3 shows different approaches that we considered for declaring local variables.

Table 8.1 summarizes all the Obsidian user studies.

8.2.1 Basic design of typestate

In order to minimize assumptions regarding how Obsidian should best represent typestate, we conducted a *natural programming* study. We focus here on two of the research questions we had:

- Are states a natural way of approaching the challenges that arise in blockchain programming?
- Which (if any) of our proposed ways of presenting states and state transitions is most understandable and usable by programmers?

These are examples of the *naturalness* research question in chapter 1 ("How can we obtain insights as to what language designs will be natural for programmers?"). We gave participants a description of a voter registration system, in which we would investigate to what extent state machines were a natural way to write smart contracts. The first task used a natural programming

Topic	Participants	Methods
Basic design of typestate (§ 8.2.1)	P1–P12	Natural programming
Fields in states (§ 8.2.2)	P20, P66–P68	Natural programming; Usability study
Permissions (§ 8.2.3)	P14–P19	Natural programming; Usability study
Typestate and ownership approaches (§ 8.2.4)	P21–P25	Usability study
Summative usability study pilots	P26–P34	Usability study
Summative usability study (§ 8.3)	P35–P40	Usability study
RCT pilots	P41–P44	RCT
RCT (chapter 11)	P45–P65	RCT

Table 8.1: Summary of all Obsidian user studies described in this paper and their participants. Participant P13 was in pilot studies. Participant numbering is consistent with prior distributed drafts, e.g., [45].

methodology: we asked participants to implement the system using pseudocode using any language features they wanted to solve the problem. Next, we gave participants a state diagram that modeled the system, and asked them to modify their pseudocode to include states and state transitions. In the third task, we gave participants a two-page Obsidian tutorial that described state blocks. However, the tutorial omitted any description of how state transitions should be written; we gave participants an Obsidian program that implemented the voter registration system but which omitted state transitions. We asked participants to fill in the missing transitions by inventing their own syntax to do so. In the fourth task, we gave participants three options for the syntax and semantics of state transitions and asked them to use each option once in an example program we provided. The first and third options are explained in Figure 8.1. An additional option involved a constructor in each state that would be invoked on transitions to that state and a rule that no code could follow a state transition.

Finally, in a fifth task, we asked participants to select one of the three options and use it to complete the voter registration program they started earlier.

We recruited a convenience sample of seven participants, most of whom were computer science undergraduates. Each participant was given a description of a program to implement and one hour to complete the implementation. We paid participants $10/hour for their time.

Only two participants invented syntax denoting states and state transitions; the rest used a conventional approach, such as an enumerated type. However, many of the approaches the remaining five participants used were unsafe, helping to justify using typestate to improve safety. For example, creating separate lists for unregistered and unregistered citizens results in the possibility of citizens appearing on both lists.

We asked six of the participants to modify their pseudocode to use states. Two created explicit state blocks with states and variables nested inside. The remaining four either maintained global state for each citizen, or gave each citizen a state field, or created empty, immutable states at the top of the program. Although the instructions forbid allowing duplicate registrations, several

```
1  contract C {
2    state Start {
3      transaction t(int x) {
4        ->S1{x1 = x};
5        toS2();
6      }
7    }
8
9    state S1 {
10     int x1;
11     transaction toS2() {
12       ->S2{x2 = x1};
13     }
14   }
15
16   state S2 {
17     int x2;
18   }
19 }
```

(a) Option 1. The dynamic state (not the lexical structure) governs which transactions may be called. For example, line 5 calls `toS2()` even though `t` is lexically in the `Start` state and `toS2()` is defined in `S1`.

```
1  contract C {
2    state Start {
3      transaction t(int x) {
4        ->S1({x1 = x})
5        if in S1 {
6          ->S2({x2 = x1})
7        }
8        if in S2 {
9          ...
10       }
11     }
12   }
13
14   state S1 {
15     int x1;
16   }
17
18   state S2 {
19     int x2;
20   }
21 }
```

(b) Option 3. Fields can only be referenced by code that *lexically* is in the state in which those fields are defined. An `if in` block can be used to enclose code that must reference fields of other states, as in line 6.

Figure 8.1: Two of the options given to participants in the *basic design* study.

participants did not check for existing registrations before processing applications.

Regarding the syntactic choices we offered in the third task, three participants preferred state constructors (part (a) in Figure 8.1), one preferred nested state blocks (part (b) in Figure 8.1), and the remaining three either did not indicate a preference or did not complete this task.

Although most of this study focused on participant *behavior*, we took the opportunity to also ask participants for their syntactic *preferences*. Five participants preferred a syntax where all the actions of a state must be lexically encapsulated in that state, as in the first alternative in Figure 8.2. Likewise, P4 felt it should not be permitted to call transactions from one state while lexically in another state: "I'm calling `S1`'s transaction from code for `Start`."

This preference led to a conflict in the design. We found through work with example programs after the study that Obsidian needed to support transactions that could be executed in several different states. For example, in the third example of Figure 8.2, line 5 may reference `balance`, even though line 5 is lexically enclosed in the `Empty` state, in which `balance` is not in scope. This represents a conflict between a syntactic preference and an expressivity concern.

Another difficulty with the constructor-based approach is that states might be entered for a variety of different reasons, requiring different code to run after the transitions. This makes the constructor-based approach likely too inflexible. These challenges underscore the importance of sufficient example-based work before conducting user studies; it is easy to design a study that provides plausible options that turn out to not support critical use cases.

This problem led us to run the study described in § 8.2.2. As a result of that study, we addressed the conflict by requiring that transactions are lexically *outside* of state declarations, like the second example in Figure 8.2. Future IDE tools could show all transactions that are possible for an object in a given state, even though their declarations are lexically outside that state's declaration.

8.2.2 Fields in states

States in contracts can have different sets of fields, so transitioning can cause some fields to exit scope and others to enter scope. For example, in Figure 6.2, the `Full` state has the `inventory` field, but the `Empty` state has no fields. This study used *natural programming* and *code understanding* methods to investigate how users users specify cleanup of old fields and initialization of new fields when invoking state transitions.

We recruited four participants, which was enough to provide substantial and useful feedback. All were Ph.D. students studying software engineering. They had an average of seven years of programming experience (ranging from three to fifteen years) and an average of 1.5 years of Java experience.

In **Part 1**, we gave participants a state transition diagram for a `Wallet` object, which could hold a license and money, and which had four states corresponding to the possible combinations of contents. Participants were also given code partially implementing the `Wallet`, with several TODO comments asking participants to invent code to add money to the `Wallet`, remove money from the `Wallet`, etc. Participants were told that the money and license should be thought of as assets, so they could not be duplicated, used more than once, or lost. The code they were given was in a language similar to Obsidian but which used some keywords that would be more familiar to a Java programmer, such as `class` instead of `contract`. As such, this was a staged natural

Transaction lexically *nested inside* state declaration:

```
contract Wallet {
   state Empty;
   state Full {
      int balance;

      // spend() is nested inside the declaration of
      // the state it belongs to.
      transaction spend(Wallet@Full this) {
        // use 'balance'...
      }
   }
}
```

Transaction lexically *outside* state declaration:

```
contract Wallet {
   state Empty;
   state Full {
      int balance;
   }
   // spend() is not nested in Full, even though it can
   // only be called in Full state
   transaction spend(Wallet@Full this) {
      // use 'balance'...
   }
}
```

Transitions when inside a state could be confusing:

```
contract Wallet {
   state Empty {
      transaction fill(int amount) {
        ->Full(balance = amount);
        // Now balance should be in scope, since new state is Full
      }
   }

   state Full {
      int balance;
      // ...
   }
}
```

Figure 8.2: Although participants preferred to have transactions nested inside state declarations (the first alternative), this desire conflicted with the need for transactions only reference fields that were in lexical scope.

programming study, since we progressively gave participants more detail about the language we were designing.

All four participants prepared assets for a state transition before making the state transition (corresponding to option (2) in Part 2 below, `S::x = a1; ->S`). Two participants felt they need to write code to handle failures during the asset preparation stage, which might lead to an improperly initialized state upon transition. One of them suggested a try-catch type wrapper for the asset preparation and transition phases.

In **Parts 2** through **4** of the study, participants were given several options. Then they were asked to implement each of the options within a given partially-implemented transaction. Finally, they were asked for their preferences.

Part 2 compared approaches for initializing fields in states during transitions. Options were:

1. Assets are assigned to fields in the transition, e.g., `->S(x = a1)` assigns the value of `a1` to field `x` of state `S`.
2. Assets are assigned to fields *before* the transition, e.g., `S::x = a1; ->S`.
3. Assets are assigned to fields *before* the transition, but the fields are in local scope even though the state has not changed yet, e.g., `x = a1; ->S`.
4. Assets are assigned to fields *after* the transition, e.g., `->S; x = a1`.

The participants successfully used all the approaches, but most of the participants preferred assigning assets to fields *before* the transition with destination state scoping (option 2). Before the study, Obsidian supported only atomic assignment (option 1, shown in Figure 6.2 on line 22). The results of these two parts motivated a language change: Obsidian now also supports option 2.

Part 3 presented two options for handling assets when transitioning from a state with an asset to a state without it:

1. The transition evaluates to a collection containing the old assets, e.g., `x = ->S` indicates that `x` is assigned the leftover assets after the transition to state `S`. If the current state is unknown statically, the contents of the collection are determined dynamically.
2. The transition evaluates to a tuple, e.g., `(x = a1) = ->S` indicates that `x` will be assigned the asset `a1` which is not present in state `S`.

There was consistent confusion about which leftover assets are assigned to option 1's collection after a transition. All participants understood the need for both options in certain cases, but would choose the tuple-like collection for more control and explicitness when the use of either approach is acceptable. We would like to implement this approach in the future but so far have not prioritized it, since the existing approach (described in Part 4, option 1), which requires that ownership of assets be surrendered before transitioning, has been effective for participants.

Part 4 focused on releasing assets owned by state fields when transitioning to states in which those fields do not exist. In contrast to *part 3*, this approach added the option of releasing assets before the transition. The choices were:

1. Assets must be released before the transition, e.g., `release(a1); ->S`.
2. The transition evaluates to a tuple of assets that are no longer owned, e.g., `a1 = ->S`.

All the participants understood the options and implemented them without mistakes. Implementing using option 2 (evaluating to a tuple) enables both approaches, so participants were asked to indicate scenarios where one option would be preferred over the other. The participants

consistently indicated that assets should be released before a transition if they are no longer needed; otherwise, they should evaluate to a tuple. This helped us prioritize our features, since releasing assets before the transition seemed to suffice.

8.2.3 Permissions: a qualitative study

Soundly enforcing typestate requires knowledge about all references to an object, which is afforded by a *permission system*. [18]. Permissions systems allow the programmer to express what a particular reference can be used for (and therefore also what it *cannot* be used for). Is there a permission system that users can understand and use effectively (a question of *naturalness*)? If so, what can we learn from users about how to design it (a question of *iteration*)? In this work, we conducted the first studies (of which we are aware) in which people other than the designers of the system were asked to use a permissions system to restrict references in a programming language. We found that our initial system design was surprisingly difficult to use, and iterated the design until it was more successful.

In order to study permissions while mitigating the *interdependency of features*, *training* and *recruiting* challenges, we extracted the permission system from Obsidian and re-cast it in Java as a set of annotation. We conducted a Wizard of Oz study where participants received documentation on a Java extension and the experimenter gave simulated compiler error messages. This approach minimized training time for participants, minimized implementation cost for ourselves, and allowed us to isolate this design decision from many others that would have otherwise distinguished the language from Java. At this point in the development of Obsidian, we assumed that it would be best to separate the notions of permissions and typestate; this approach was reflected in the study materials but may surprise a reader who has studied Figure 6.2, which reflects the final Obsidian version, which combines the two. The training materials explained the annotations: `@Asset`, which applied to classes; and `@Owned`, `@Shared` (no restrictions but could not co-exist with typestate-specifying references), and `@ReadOnlyState` (restricting state modification), which applied to references. We recruited six participants (P14–P19), which was enough to provide substantial and useful feedback. They had a mean six years of programming experience (ranging from three to nine years) and a mean two years of Java experience.

The study included five parts. Since our goal was to identify as many usability problems as possible, we revised the design and instructions after each participant. The first three participants were given 1.5 hours to do the first four parts; the last three were given two hours to fit in a fifth part of the study. An experimenter was available to answer questions.

Part 1. To motivate the need for language features to prevent bugs, we gave participants a 163-line Java medical records system and asked the first two participants to find a bug in which a patient could refill the prescription more times than specified. The first participant did not find the bug within 30 minutes; the second did so just as time expired. To conserve time, we gave the other participants five minutes to inspect the code and explained the problem to them.

We conclude that at least some programmers who use traditional languages would have difficulty detecting the kind of bug that Obsidian prevents. This provides further evidence that if users use Obsidian, the compiler will help them detect bugs that otherwise might go undetected.

Part 2. We told participants we would prevent the previous bug by distinguishing between two kinds of references. "Considering an object *o*: *Kind #1*: There is only one reference of kind

#1 to *o* at a time. *Kind #2*: There may be many references of kind #2 to *o* at a time." We asked participants to propose names for the two kinds of references. Note the careful language avoiding bias toward specific vocabulary. Participants' name suggestions included:

Kind #1: KeyReference, UniqueReference, Owned, Singleton reference, Resource handle, @default

Kind #2: DuplicateReference, ForeignKeyReference, Borrowed, Flyweight pattern reference, const pointer

The results were too inconsistent to justify an particular choice in the language; all the suggestions were distinct, and some of them were not appropriate in context (*unsound proposals* challenge). Obsidian uses `Owned`, which is at least consistent with one suggestion, and `Unowned`.

Part 3. To evaluate the usability of ownership, we gave participants an ownership tutorial and told them we had chosen [*no annotation*, `@ReadOnly`] (first participant) or [`@Owned`, *no annotation*] (later participants) as keywords. We asked them to modify the code from Part 1 to fix the bug. We hoped participants would require that `Prescriptions` deposited in a `Pharmacy` be owned and that the `Pharmacy` take ownership; thus, a deposited `Prescription` could not be deposited in a second `Pharmacy`. Completion times ranged from 3 minutes to 40 minutes (*variance* challenge). Two participants did not finish, one of whom we stopped after 38 minutes to prioritize other tasks.

We were surprised that many of the participants found this task very difficult. We expanded the tutorial to include a practice section for later participants. In general, participants were not prepared to use a type system to address a bug that they thought of in a dynamic way. For example, P16 wrote `if (@Owned prescription)`, attempting to indicate a dynamic check of ownership. We asked participants who wanted to use dynamic approaches for enforcement to use the language feature instead. P14 commented "I haven't seen... types that complex in an actual language ... enforced at compile time."

P17 had trouble guessing what the compiler could know, expecting an interprocedural analysis (which would be non-modular). For example, in a case where an owned object was being consumed twice, P17 expected the compiler to give an error on the second `spend` invocation. Instead, because the second invocation was inside a helper method, the compiler reported the error on the invocation to the helper method, which took an owned argument and invoked the second `spend`.

P17, P18, and P19 had difficulty determining which variables should be annotated `@Owned`. In one case, a lookup method took an object to search for, but P17 specified that it should take an owned reference. Then he was stuck after invoking it: "How can I get the annotation back?" But this was impossible except via adding another method, since he had already given away ownership. Likewise P17 was confused by whether accessors should return owned references. Mistakes could be costly. For example, P19 unnecessarily annotated as `@Owned` a class that was contained in a collection, which caused a problem iterating through the collection. He made the reference to the current list element `@Owned`, which would require removing each item from the collection when iterating over it in code that was not supposed to modify the container at all.

Parameter-passing and assignment were common points of confusion. P18 asked what happens when passing an `@Owned` object to a method with an unowned formal parameter (ownership was not passed in this case). P19 said, "when I [annotate this constructor type `@Owned`], I'm

not sure if I'm making a variable owned or I'm transferring ownership." P17 was surprised that assignment from an owned reference to an unowned-type variable did not transfer ownership. We later addressed this confusion by making assignment always transfer ownership; participants in later studies were generally not confused about which assignments transfer ownership.

From this portion of the study, we came to two general conclusions. First, the semantics of ownership needed to be as explicit and as simple as possible. This likely generalizes to many different kinds of complex language constructs: implicit behavior, although sometimes convenient for experts, can be baffling to novices. When the behavior can be made explicit without making the language inconvenient for experts, that should be done. Second, language design decisions that have structural implications (as is the case for ownership) require substantial high-level training; we refined the training materials in future studies to give more explanation and examples.

Part 4 introduced the notion of *assets*. After a tutorial explaining the properties of assets, participants were asked to invent code that could indicate a particular owned reference was *intentionally* going out of scope. Two participants suggested `@Disown` and `free` to abandon owned references; the rest did not have time to answer or had no suggestions. We chose `disown` for Obsidian, since `free` has additional memory management connotations that are not relevant here.

Part 5 introduced typestate, starting with the fourth participant. Participants read 2.5 pages on typestate in Obsidian (as it existed then), including `@ReadOnlyState`, `@Shared`, and `@Borrowed` (which was for temporary ownership transfer in invocations). Ownership was the default, so no `@Owned` was needed. The tutorial also explained `available in` and `ends in`, which at the time specified state assumptions and guarantees for methods (before we changed to using `this` parameters instead, e.g., as on lines 19 and 27 of Figure 6.2). Then, they were asked to annotate uses of `Bond` in a 212-line Java program implementing a financial market. They were told to use ownership and state specifications whenever possible.

Consistent with Part 3, some participants were more comfortable with a dynamic perspective on ownership rather than a static one. P18 felt that `ends in` declarations were redundant with the transition code already in the method implementations, but these declarations allow separation of interface and implementation and modular checking. P19 wanted to use borrowing to represent the notion that the `BondMarket` owns a `Bond`, but an `Investor` borrows it for a while. In fact, borrowing was only appropriate for the duration of a method invocation. We later changed the design of the formal parameter syntax to remove the need for `@Borrowed`; now, if no ownership change is specified (via the >> operator), ownership remains unchanged.

P19 required significant prompting by the experimenter to make maximum use of typestate. First, P19 added annotations on methods but not on any variables. After prompting, he added dynamic checks in one place but required prompting to add static typestate specifications. This suggests that tools may be needed to help users obtain the most benefits from the language. On the other hand, P18 specified `@Asset` on `Bond` without being asked to do so, explaining "because it's something important and I don't want to get it out of scope…"

Overall, understanding the limitations of the type system and compiler may be an obstacle for some people. Users will need training to reason about what typestate can do, but the observations above motivated language changes that simplified the design without lowering the expressivity or safety. Tools could mitigate the limitations of traditional type systems by providing sophisticated static analyses rather than taking a traditional type checking approach (as Obsidian does), and by

providing detailed, explanatory errors.

8.2.4 Comparing typestate and ownership approaches

We were interested in evaluating a new approach we invented, which was motivated by the confusion we observed in the prior study (in part a question of *naturalness* and in part a challenge of *training*). We invented a new approach: fuse the notions of ownership and typestate in order to simplify the type system, and the next study refined this design. This design has the benefit of eliminating `Shared` references that also specify typestate, which would then have to be disallowed to preserve soundness. Thus, the type `Bond@S` is always implicitly an owned reference for any state `S`, and users can write any permission instead of `S`, as in `Bond@Unowned`.

We were also interested in another usability concern. Consider *Approach 1* in Figure 8.3. A reader of line 1 might expect that the type of `bond` would always be `Bond@Offered`. In fact, after line 2, the type is `Bond@Sold` due to the call to `buy`. A fundamental aspect of Obsidian is that ownership can change, so if a variable declaration includes any ownership information, the variable's ownership status may later be inconsistent with its declaration.

We initially invented two possible solutions to address this problem. One idea for addressing this involved incorporating types into variable names, shown in *Approach 2*. The annotations pertain to the *current* type rather than the *new* type. The reader would have to look at only the most recent operation to infer the new type of a variable rather than having to potentially read the whole sequence since the declaration.

Approach 3 represents another idea: adding static assertions. Line 3 shows a static assertion that `bond` references an object in state `Sold`, which serves as documentation. Unlike traditional assertions, however, the compiler checks correctness. The intent is to make it easier for programmers to determine the types of variables.

We conducted studies with participants in the first three conditions. Inspired by observations of those participants, we invented *approach 4*. This approach is like approach 3 except that it *removes* state specifications from local variable declarations. The removal was not part of the original design but was inspired by early results of this study.

Participants

We required that participants be familiar with Java and we administered a simple Java pre-test. We recruited five students (P21–P25). Based on self-reports, they had an average of about four years of Java experience (ranging from one to ten years) and an average of one year of professional (paid) software development experience (ranging from zero to three years).

Procedure

Participants spent between 1 and 1.5 hours on the study. We used a Qualtrics survey to ask participants a series of questions regarding Obsidian programs, but the study took place in a lab and an experimenter was available to answer questions. The survey both taught aspects of the language and provided an opportunity for assessment. Most of the questions were typical *code understanding* questions, which gave snippets of code and asked whether the compiler would give

Approach 1: traditional declarations

```
1 Bond@Offered bond = new Bond();
2 bond.buy(...);
```

Approach 2: types in variable names

```
1 Bond bond@Offered = new Bond();
2 bond@Offered.buy(...);
```

Approach 3: static assertions

```
1 Bond@Offered bond = new Bond();
2 bond.buy(...);
3 [bond@Sold];
```

Approach 4: no states in local variable declarations

```
1 Bond bond = new Bond();
2 bond.buy(...);
3 [bond@Sold];
```

Figure 8.3: Variable declaration approaches

an error or what the code meant. We assigned participants to one of the four conditions above according to what we hoped to learn from each trial: approach 1 for P22, approach 2 for P21, approach 3 for P23 and P24, and approach 4 for P25.

Results and Discussion

P22, who was given approach 1 (with permissions and states specified only in declarations), tried to guess the compiler's behavior, saying things like "If the compiler was smart. . . ". For example, P22 expected that the language would infer an implicit `@Off` in the declaration `LightSwitch s1 = new LightSwitch()`. P22 also expected that although changes of state were permitted via transactions, state-mismatching assignment to variables would be forbidden, even though approach 1 assumes that states can be inconsistent with type declarations. This approach would be inconsistent and P22's confusion suggests that the type-declaration approach is problematic.

Including types in variables names seemed to be confusing as well. P21 expected that ownership was *not* passed into method calls even when an owned reference was passed. P21 was also surprised that no ownership annotation meant that there was no ownership, instead expecting this to mean that ownership was unknown.

Participants in condition 3 seemed to do much better. For example, although the materials did not use the word *assertion*, P23 observed that the annotations were assertions. P23 liked the system, commenting "Perfect, I like this, this is very nice. I wish Java had this; it would have saved me a lot of bugs." As we obtained additional confidence in the value of approach #3, we added additional material. For P24, we changed assertions to use `@` rather than the initial >> so that we could use >> to specify type changes in transaction parameters. With P25, we used `?` to indicate lack of static state knowledge. We later simplified the system because this approach

Q14.6.

```
resource class Money {…}
resource class Bank {
  Money@Owned myMoney;

  void depositMoney(Money@Owned >> Unowned deposit) { … }
  Money@Owned withdrawMoney() {…} // Withdraws all money
}

class Test {
   Bank b;

  void putMoneyInBank(Money@Owned >> Unowned m) {
    // At the beginning, m owns an instance of Money.
    Money q = m; [m @ Unowned] // Location (A)
    b.depositMoney(q); [q @ Unowned] // Location (B)
    // Location (C)
    b.depositMoney(b.withdrawMoney()) // Location (D)
  }
}
```

At location (A), w hat happens?

- (•) Ownership is transferred from m to q
- ( ) Compiler error: q needs to acquire ownership, but this line does not transfer it
- ( ) This line is correct by itself, but will cause an error at Location (B) because q is not actually owned
- ( ) Something else (explain what) [ ]

Figure 8.4: An example question assessing understanding of ownership transfer. The correct answer is selected, since assignment transfers ownership.

was ambiguous, leaving notations `Owned`, `Unowned`, `Shared`, and unions of specific states (separated with `|`).

P24 was confused because state specifications on local variables were redundant. For example, in `LightSwitch@Off s = new LightSwitch()`, the `@Off` portion is redundant because the compiler already knows the state of the new object due to the constructor's declaration. To resolve this, we added approach 4, removing typestate and permission annotations from local variable declarations; in contrast, permissions are always specified for fields and formal parameters. In those cases, the annotations are important because they constrain types of variables at the end and beginning of transactions.

In summary, this study motivated the removal of state specifications from local variable declarations and provided initial evidence that static assertions are likely to be a convenient way for programmers to specify states and permissions of local variables. We also obtained evidence that with these other changes, static state assertions are understandable by current Java users with little extra training.

8.2.5 Threats to validity

The studies share common threats to validity, many of which correspond to the *external validity* challenge: our participants may not be representative of the population of blockchain programmers; we had limited numbers of participants in each trial; and our tasks may not reflect the reality of blockchain programming. We believe, however, that the population of likely language users is *more* skilled than our participant population, which mostly consisted of students, so if the students are successful in completing tasks, that aspect of the result is likely to generalize. We did not seek to identify all possible usability problems, but rather to identify the most common and severe ones associated with particular design decisions so that we could try to address them. Because there were so many different design decisions, we focused on those for which we had prior evidence that there might be usability problems.

8.3 First usability study of Obsidian

In order to assess whether our changes to Obsidian had resulted in a language in which programmers could be effective, we designed a summative usability study. We gave the participants the complete Obsidian language, including its compiler, and asked them to complete three programming tasks. We were interested in whether the participants experience the same usability problems as the prior participants and whether there were sufficiently serious usability problems left to prevent them from completing their tasks. We found that given enough time, most of the participants were effective at completing the tasks we gave them.

The design of the study was informed by several of our methodological contributions (chapter 12), and thus also served to assess their value. We trained the participants in Obsidian as part of the study; we recruited local students to participate; we used multiple programming tasks, rather than one long one; and we we managed task times according to our guidelines.

8.3.1 Participants

We solicited experienced Java programmers to take a short screening test online, which took an average of about 9 minutes to complete. We accepted into the three-hour study only those who answered at least five of six basic Java questions correctly, including one question about aliasing. Of 18 completed surveys, 11 people met our screening criteria. We got six participants (P35-P40), whom we compensated with $50 Amazon gift cards. The participants had an average of 9 years of programming experience, 2 years of professional experience, and 2 years of Java experience. One self-identified as female; the rest identified as male.

8.3.2 Procedure

The previous studies focused on particular aspects of the design, in many cases by giving participants languages that were not precisely Obsidian. To evaluate our final design, we conducted a usability evaluation. Because Obsidian provides stronger safety guarantees than existing languages such as Solidity, and because of our prior experience showing that it would be very challenging to develop a linear type system that would be usable at all, we focused our work on whether people could effectively complete tasks, not on whether people could complete tasks faster than in existing languages. We based our work on pilot studies that we previously conducted [104].

The experimenter gave low-level guidance, such as how to invoke the compiler. Further, the experimenter provided assistance that simulated more mature tools. For example, when a participant attempted to debug an error that was reported on line 38 by examining line 33, the experimenter pointed out the discrepancy, since the IDE we provided did not highlight the appropriate line.

After completing the tutorial, which included seven programming exercises, we gave participants starter code for the three main tasks, described below. Although participants used the compiler, they were not given tests or a runtime environment, since the focus of our usability study was the type system (recall that Obsidian is designed to detect as many bugs as possible at compile time, since runtime detection may be too late to ensure safety). Although the first two tasks were short in order to reduce variance, we allowed the third task to be more open-ended to see whether participants would be able to complete a more challenging task.

The first task, *Auction*, simulated an English auction, in which bids are public, and the bidder who offers the highest price must pay that price for the item. We added the additional constraint that bids were required to come with `Money` so that bids could be guaranteed to be viable (a bidder could not issue a bid and then fail to pay for the item). As a starter task, we asked participants to finish implementing `createBid`, requiring them to invoke a constructor. They also needed to finish implementing `makeBid`, which records a new bid from a client. In `makeBid`, we were interested in whether they initially wrote code that accidentally lost the previous bid, which held the associated `Money` (before receiving a compiler error), indicating that Obsidian's typechecker had helped them avoid losing track of an asset.

The second task, *Prescription*, corresponded to the medical records system in the Permissions study section (§ 8.2.3); we were interested in whether our improvements enabled participants to reason more effectively about the code than we had observed in the previous studies. We asked participants to fill in the type signature for the `consumeRefill` and `depositPrescription`

	Task completion times (hours:minutes)			
	Tutorial	**Auction**	**Prescription**	**Casino**
P35	1:31	0:13	0:18	1:01
P36	2:12	0:28	N/A	N/A
P37	1:03	0:33	0:46	0:36*
P38	2:18	0:46	N/A	N/A
P39	1:14	0:22	0:27	0:51*
P40	1:11	0:12	0:22	0:58

Table 8.2: Usability test results. * indicates insufficient time to finish the task. N/A indicates insufficient time to start the task.

transactions, as well as completing the implementation of `fillPrescription`.

The *Casino* task was more open-ended and included directions and requirements for what operations should be supported, as well as low-level starter code, such as implementations of `Money` and `Bet`. It asked participants to implement a `Casino` that takes bets on games. When games are complete, the casino enables winners to collect their winnings. We were primarily interested in participants' abilities to reason about ownership and typestate and to design architectures that could effectively use ownership.

8.3.3 Results and Discussion

Results for the tasks are summarized in Table 8.2. With P38, to assess to what extent the tutorial materials stood alone, the experimenter declined to answer Obsidian-related and debugging-related questions. However, this made the first task perhaps unrealistically difficult and lengthy, resulting in insufficient time for the other tasks.

In the **Auction** exercise, two of the six participants accidentally introduced a bug in which an asset was lost: they overwrote `maxBid`, which held money. The compiler gave an error message and they corrected their mistake, but if they had been using Solidity, its compiler would not have caught the bug. After P36, we slightly simplified the Auction exercise by removing a subtask and refactoring to inline a TODO that had been put in a helper transaction. The above times are adjusted to remove the extra time P35 and P36 spent on the removed task (1 and 8 minutes, respectively).

Some participants seemed to think carefully about ownership and wrote the correct code quickly. Others seemed to focus on satisfying the compiler, and their work took longer. For example, P38 got an error message after overwriting the owned `maxBid` reference, and "fixed" it with `disown`. This choice may be a result of weaker programming skills and lack of help in the tutorial; P38 took the longest on the tutorial, and was surprised to not be given a design diagram for the (< 300-line) Auction starter code. We changed the tutorial to emphasize that `disown` should be used to throw away assets.

In the **Prescription** task, as with other tasks, variance was large. For example, one reason for P38's long completion time was that P38 had used Python most recently and, despite the tutorial, sometimes wrote Python-like syntax, which did not parse (one example took four minutes to fix).

At the time, we were hoping that participants would be able to complete the tasks entirely on their own, but in retrospect, we may obtain *more relevant* results by carefully providing appropriate help (which we provided to all the other participants).

We were interested in participants' ability to reason effectively about ownership. All of the participants who started Prescription were able to complete it. P37 encountered some difficulties due to shortcomings in Obsidian's support for dynamic state tests. Currently, Obsidian does not allow dynamic state tests to be used as arbitrary Boolean expressions, e.g., `if (x in S && e)` where `e` is an arbitrary Boolean expression. Likewise, `if (x not is Owned)` is not supported (perhaps this was inspired by Python's `is` operator). In the latter case, P37 developed some intuition: "Ownership doesn't feel like something I should be using in this way..." and restructured the code to check `if (maybeRecord in Full)`, which was correct. In another case, the compiler found a bug in which the code assumed that a collection must contain an element, a benefit of not allowing `null` in the language.

The *Casino* task was substantially more open-ended than the other tasks, requiring substantially more time, but participants who had a full hour for the task were able to finish it. Some participants defined states in the `Casino` contract (P35, P39), whereas others relied only on the states in the Game contract (P37, P40). Both approaches led to a lot of dynamic state tests, since the Casino object had to check to make sure the Game object was in an appropriate state. These checks could have been avoided if the different states of Casino had different typestate specifications for their references to the Game, an idea that occurred to P40 in retrospect. This observation represents an opportunity for a future version of Obsidian in which states of owning objects are coupled to states of owned objects, reducing the need for dynamic checks.

We noticed that participants who did better on the "advanced Java" portion of our screening test seemed to complete tasks faster. We found that those test scores were negatively correlated with completion times of the Auction task ($r(4) = -.96$, $p < .01$). We regard this result as tentative because there were minor differences among the trials; we defer a definitive conclusion to a future quantitative study. The correlation between the scores and the tutorial completion times was not significant, likely an artifact of the small number of participants. This suggests that much of the variance (91%) in Auction completion times is explained by prior programming background. We hypothesize, then, that participants who have sufficient OOP understanding can learn Obsidian and use the language effectively in only about 90 minutes.

8.4 Conclusion

Our initial prototype, which was based on prior typestate-based programming languages, presented significant usability challenges when we evaluated it in formative studies with users, but we were able to leverage those studies to improve the design. Most of the participants in a summative usability study of our revised language were able to use the language effectively for small tasks. This is surprising, considering that we did not have to weaken the safety properties that the language provided. We regard this as evidence that usability methods can be used to develop languages that help programmers be effective at obtaining stronger safety guarantees than they might otherwise be able to obtain.

Chapter 9

Detailed Design and Formal Aspects of Obsidian of Obsidian[1]

9.1 Introduction

This chapter describes the Obsidian language formally. Obsidian rests on a formal foundation, Silica; this chapter describes Silica, defines Obsidian by translation to Silica, and gives details regarding Obsidian's implementation and architecture.

This chapter makes the following contributions:

1. It shows 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. It shows 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. It describes Silica, the core calculus that underlies Obsidian. It proves 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.

§ 9.2 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 § 9.3. § 9.4 describes Silica, the core calculus underlying Obsidian, and its proof of soundness (although the proof itself is in appendix A.3). The syntax of Obsidian is specified in § 9.5 as an extension and modification of the syntax of Silica, and the semantics are defined by translation to Silica.

[1] Portions of this chapter previously appeared in [48].

9.2 Obsidian language design

9.2.1 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 [40]. Traditional declarations are also typical in prior typestate-supporting languages, such as Plaid [198]. 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 [138] 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.

Static assertions alleviate this problem. 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 run 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.

9.2.2 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 6.2). 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 [138] and results by Stylos et al. [195], 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.

In order 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.

9.2.3 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 [198] 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. [11] 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`.

9.2.4 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 9.1.

```
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 }
```

Figure 9.1: Obsidian's approach for handling transitions.

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, how is the programmer supposed to implement `swap`? One possibility is to add another state, as shown in Figure 9.2.

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

```
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 }
```

Figure 9.2: An alternative approach for handling transitions.

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 [198] addressed the problem by not including type names in fields, but that approach may hamper code understandability [40].

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. In order 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:

```
1 contract AContract {
2     state S1;
3     state S2;
4
5     AContract@S1 c;
6     private (AContract@S2 >> S1 c) transaction t1() {...}
7 }
```

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 [55].

9.2.5 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 Fig. 9.3. 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 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, 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, 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, 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

```
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 }
```

Figure 9.3: A dynamic state check example.

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 run 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 run 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 run 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, 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 [174].

The dynamic state check mechanism is related to the *focusing* mechanism of Fahndrich and DeLine [75]. 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.

9.2.6 Parametric Polymorphism

Parametric polymorphism is particularly important for Obsidian in order 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`.

9.3 System design and implementation

Our current implementation of Obsidian supports Hyperledger Fabric [200], 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 Fig. 9.4.

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, the compiler reports an error.

9.3.1 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* [79] 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 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

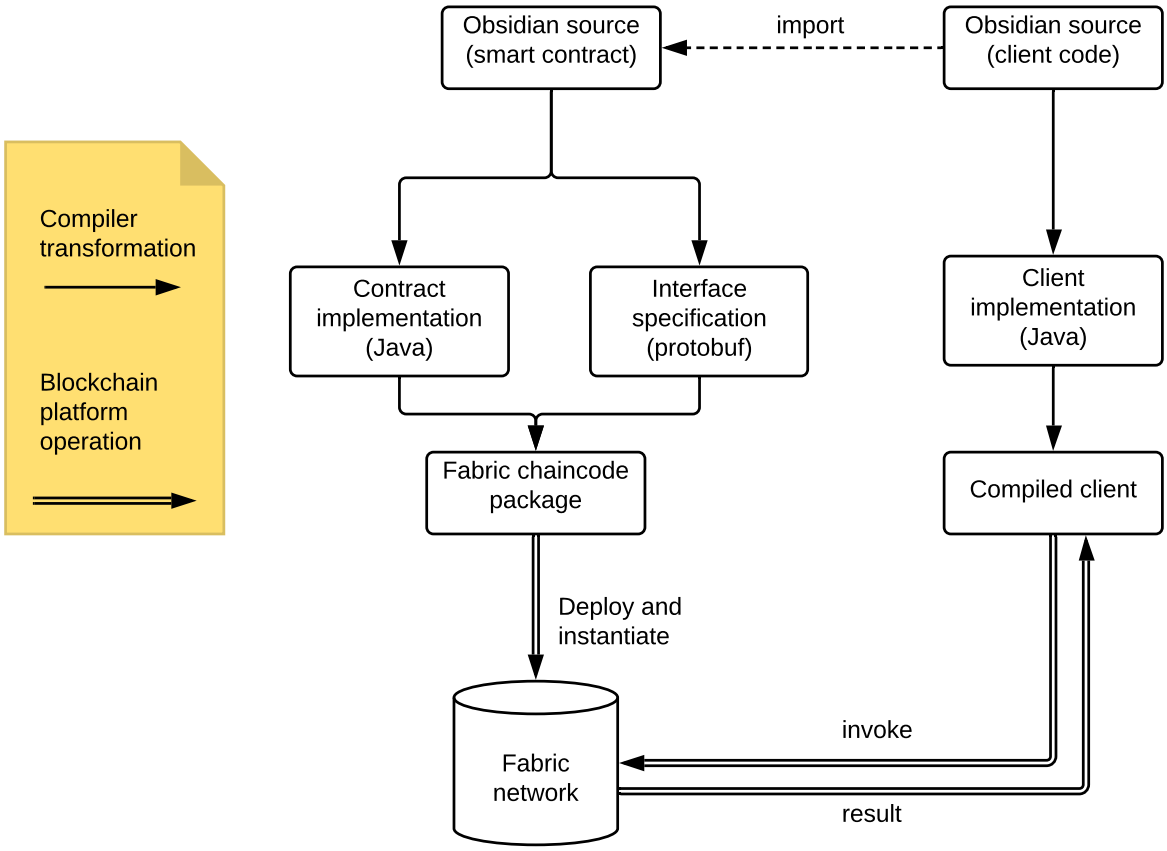

Figure 9.4: Obsidian system architecture

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.

9.3.2 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, 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 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 in order 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. Fig. 9.5 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   // suppose t1 requires that c references an object in state S1
3   c.t1();
4 }
```

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* [114], 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.

```
1  import "TinyVendingMachine.obs"
2
3  main contract TinyVendingMachineClient {
4      transaction main(remote TinyVendingMachine@Shared machine) {
5          restock(machine);
6
7          if (machine in Full) {
8              Coin c = new Coin();
9              remote Candy candy = machine.buy(c);
10             eat(candy);
11         }
12     }
13
14     private transaction restock(remote TinyVendingMachine@Shared machine) {
15         if (machine in Empty) {
16             Candy candy = new Candy();
17             machine.restock(candy);
18         }
19     }
20
21     private transaction eat(remote Candy @ Owned >> Unowned c) {
22         disown c;
23     }
24 }
```

Figure 9.5: 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 Fig. 6.2) to have Shared receivers, since top-level objects are never owned by clients.

9.3.3 Ensuring safety with untrusted clients

If a client program is written in a language other than Obsidian, 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.

9.4 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 percent) [68]. Silica is designed in the style of Featherweight Typestate [78], which is itself designed in the style of Featherweight Java [100]. Silica leverages key concepts and notation, such as *type splitting*, from Featherweight Typestate. However, Silica differs significantly from Featherweight Typestate (FT); the differences are described in more detail in Table 9.1.

Fig. 9.6 shows the syntax of Silica. Silica uses A-normal form [174] 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 [100], 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.

$C \in$ 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)
	|	s.$m\langle\overline{T}\rangle(\overline{\text{x}})$	
	|	let $x : T = e$ in e	
	|	new $C\langle\overline{T}\rangle$@S($\overline{\text{s}}$)	(contract fields, then state fields)
	|	s $\rightarrow_{\text{Owned\|Shared}}$ 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)

Figure 9.6: Abstract syntax of Silica.

9.4.1 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*.

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 9.6. Γ is used to track the typing constraints on type variables.

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

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

$T_1 \Rrightarrow T_2/T_3$ **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. 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.

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

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

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

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

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 ($T_1 \Rrightarrow T_2/T_3$) (9.4.1), 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.

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

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).

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

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.

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

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.

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

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.

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

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

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

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

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, 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 ($\Gamma \vdash \textit{bound}\,(T_C@T_{ST})$ 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 $specializeTrans_\Gamma\left(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle\right)$ (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.

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

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.

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

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.

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

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.

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

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

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

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 $\mathsf{if}\ x\ \mathsf{in}_{\mathit{shared}}\ \overline{S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, 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, 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.

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

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*).

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

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

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.

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

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.

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

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.

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

$\boxed{M\ \textbf{ok in}\ C}$ **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.

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

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).

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

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.

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

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.

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

IFACE **ok** **Well-typed Interface**

A well-typed interface must have well-formed type parameters (as defined by *genericsOk* in AJ:30).

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

PG **ok** **Well-typed Program**

A well-typed program consists of well-typed contracts, well-typed interfaces, and a well-typed expression.

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

9.4.2 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.

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

$$\frac{}{\Gamma \vdash \mathsf{unit} <: \mathsf{unit}}\ ¡:\text{-UNIT} \qquad \frac{\Gamma \vdash T_{ST} <:_* T'_{ST}}{\Gamma \vdash T_C@T_{ST} <: T_C@T'_{ST}}\ ¡:\text{-MATCHING-DEFS}$$

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

$$\frac{\Gamma \vdash T_{ST} <:_* T'_{ST} \quad \mathit{def}(C) = \mathsf{contract}\ C\langle\overline{T_G}\rangle\ \mathsf{implements}\ I\langle\overline{T'}\rangle\{\ldots\}}{\Gamma \vdash C\langle\overline{T}\rangle@T_{ST} <: \sigma\left(\overline{T/T_G}\right)\left(I\langle\overline{T'}\rangle@T'_{ST}\right)}\ ¡:\text{-IMPLEMENTS-INTERFACE}$$

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

$\boxed{\Gamma \vdash T_{ST1} <:_* T_{ST2}}$ **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.

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

$$\frac{\Gamma \vdash \mathit{bound}_*\,(p) = T_{ST}}{\Gamma \vdash p <:_* T_{ST}}\ <:_*\text{-VAR} \qquad \frac{\overline{S} \subseteq \overline{S'}}{\Gamma \vdash \overline{S} <:_* \overline{S'}}\ <:_*\text{-S-S'} \qquad \frac{}{\overline{S} <:_* \mathsf{Owned}}\ <:_*\text{-S-O}$$

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

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

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

9.4.3 Silica Dynamic Semantics

In order 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 [78], 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.

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

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

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

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

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

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

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, etc. 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.

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

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

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

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

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

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

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

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.

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

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

The two rules above use $lookup_\xi$:

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

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

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.

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

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

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

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

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

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, that would generate multiple distinct temporary owning references.

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

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

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

$\boxed{e}_o$ 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.

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

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

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

9.4.4 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 A.3. 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 [78].

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 $()$:

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

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

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

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.

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

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

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 in order 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

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

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

$$\frac{}{T_C@Unowned \leftrightarrow T_C@Unowned}\ \textsc{UUCompat}$$

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

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

$refTypes$ 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 9.4.1 ($<^l$)**.** *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 9.4.0.1 ($<^l$-reflexivity)**.** *For all Δ, Γ, Σ, $\Delta <^l_{\Gamma,\Sigma'} \Delta$.*

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

Lemma 9.4.1 (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 $T = \mathsf{unit}$, then $v = ()$.*

Proof. By inspection of the typing rules. □

Lemma 9.4.2 (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 A.3.1. □

Theorem 9.4.1 (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**, $\Sigma, e \to \Sigma', e'$ 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, $e = \mathbb{E}[\textsf{if } s \textsf{ in}_{shared} T_{ST} \textsf{ then } e_1 \textsf{ else } e_2]$ 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 A.3. □

Theorem 9.4.2 (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', $\Gamma'; \Delta' \vdash_s e' : T' \dashv \Delta'''$, $\Gamma', \Sigma', \Delta'$ **ok**, and $\Delta''' <^{l}_{\Gamma,\Sigma'} \Delta''$.*

Proof. The proof, which proceeds by induction on the dynamic semantics, can be found in Appendix A.3. □

Because invocation results in additional indirect references being added to the context, which may be used in e', we need to typecheck e' in a new typing context. Informally, the guarantee is that all indirect references in the resulting context are at least as specific as the ones that were given in the initial typing judgment.

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 9.4.3 (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* $e = \mathbb{E}[\textsf{disown } s]$, *where* $\rho(s) = o$.

Proof. The proof, which proceeds by induction on the typing derivation, can be found in Appendix A.3. □

9.5 Obsidian Language Definition

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

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 9.1 shows the differences and how the translator handles them.

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 [154].
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 [154].
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.

Table 9.1: Differences between Obsidian and Silica

9.5.1 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 run time 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 9.6. 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.

9.6 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.

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

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

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

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.

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

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.

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

9.7 Conclusion

With Obsidian and Silica 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.

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.

Chapter 10

Obsidian Case Study Evaluation[1]

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, 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.

[1] Portions of this chapter previously appeared in [48].

10.1 Case study 1: Parametric Insurance

10.1.1 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?

In order 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.

10.1.2 Requirements

The main users of the insurance system are **farmers**, **insurers**, and **banks**. Banks are necessary in order 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.

10.1.3 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 in order 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, the farmer submits the appropriate payment.

Later, if a weather event occurs that would justify filing a claim, 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.

10.1.4 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 Fig. 10.1. All of the objects reside in the blockchain except as noted. The full code for this case study is in appendix C and online [41].

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 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

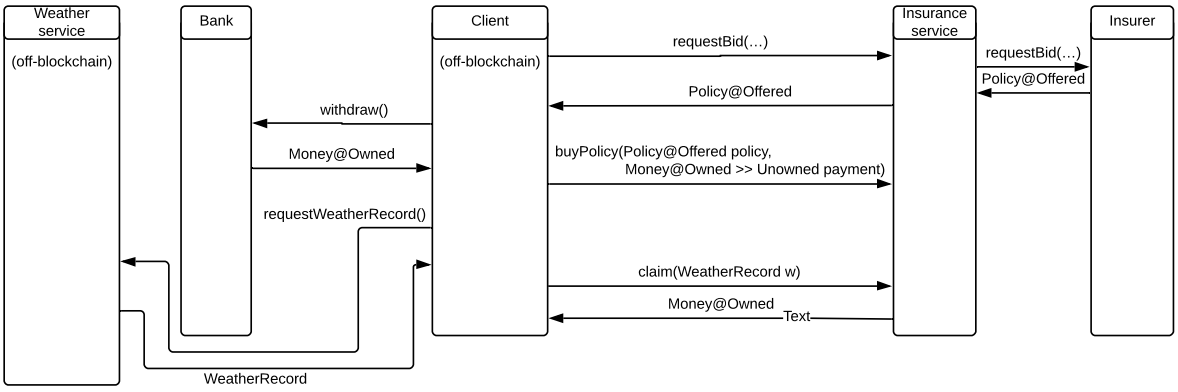

Figure 10.1: Invocations sent and results returned in a typical successful bid/claim scenario.

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, 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.

It is instructive to compare the Obsidian implementation to a partial Solidity implementation, which we wrote for comparison purposes. Figure 10.2 shows an example of why parts of the Obsidian implementation are substantially shorter. Note how the Solidity implementation requires repeated run 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 between farms and weather events).

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.

10.2 Case study 2: Shipping

10.2.1 Motivation

Supply chain tracking is one of the commonly-proposed applications for blockchains [99]. 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

```
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.

Figure 10.2: Comparison between Obsidian and Solidity implementations of a Policy contract from the insurance case study.

was done.

10.2.2 Results

The final implementation, which is in appendix C, consists of 141 non-comment, non-whitespace, non-printing lines of Obsidian code [41]. 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.

Fig. 10.3 summarizes an early design of the Shipping application[2]. 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

[2]This version corresponds with git commit 8106e406e8ca005f8878dea5ac78e54b439fe509 in the Shipping repository.

become Shared, not Owned, which is inconsistent with the type of `depart`, which requires that `this` be owned (and specifically in state `InTransport`).

Fig. 10.4 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 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 (Fig. 10.3), `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* [123]. 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 in order to improve the design. For example, the first error message in the initial version of the application shown in Fig. 10.3 is: `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 [41], which required 197 non-comment, non-whitespace lines, so the Obsidian version (141 lines) took 72% as many lines as the Solidity version. The implementation is also included in appendix C. The translation was straightforward; we translated each state precondition to a run 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 run time checks of properties that were established statically in the Obsidian version.

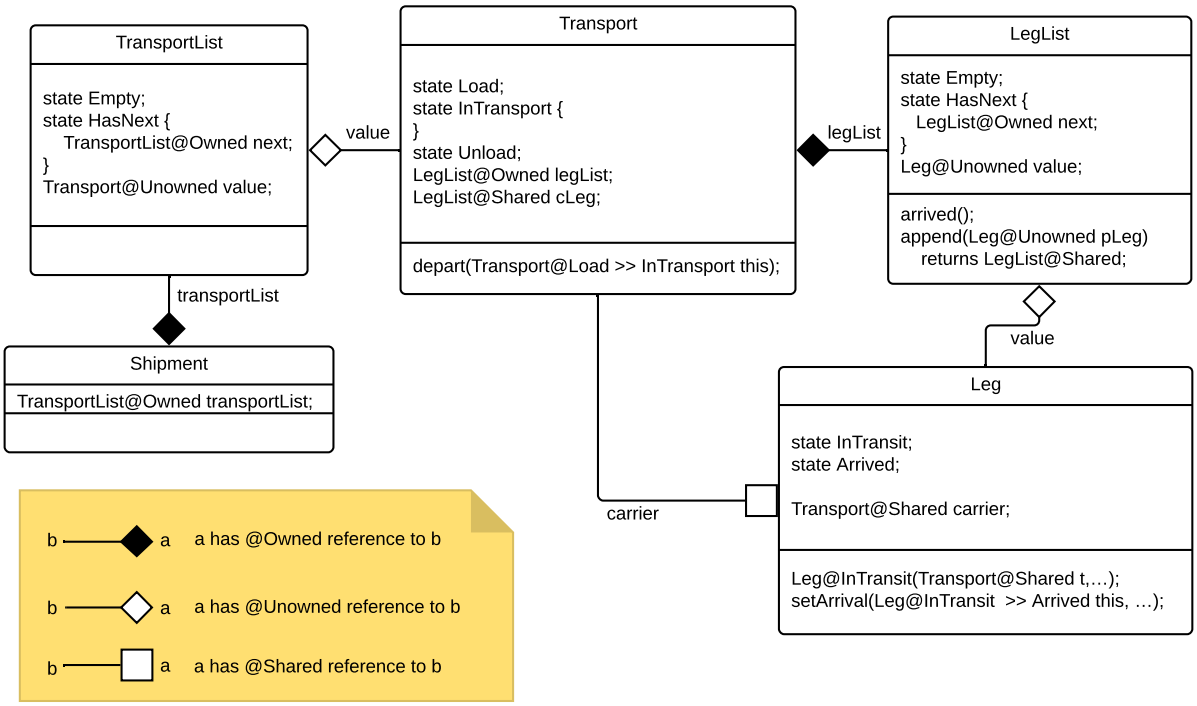

Figure 10.3: Initial design of the Shipping application (which does not compile).

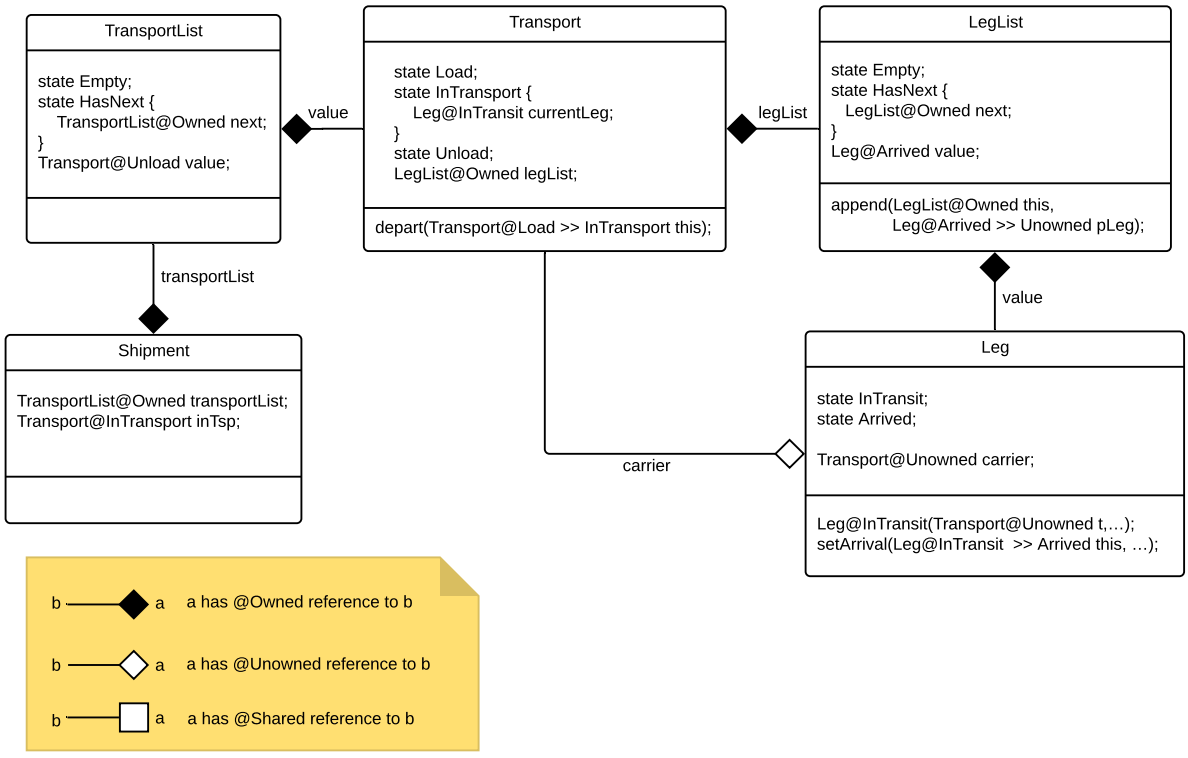

Figure 10.4: Revised design of the Shipping application.

10.3 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.

Chapter 11

An Empirical Study of Ownership, Typestate, and Assets in Obsidian[1]

11.1 Introduction

In our small usability study, described in § 8.3, most of our participants could complete particular programming tasks, given enough time. However, we wanted to assess whether Obsidian was an improvement over Solidity, since Solidity was the language that was most commonly used for writing public smart contracts. Very few other languages (for example, HANDS [146] and Quorum [191]) have been evaluated empirically in a summative evaluation, and those had relatively conventional type systems. Could we train participants to use Obsidian in a short time, given the unfamiliar aspects of its type system, and then show that they could use it more effectively than Solidity? If so, the techniques used to teach Obsidian could potentially be applied in other contexts to enable more people to obtain stronger safety guarantees through strong type systems.

This chapter includes the following contributions:

1. Showing how to conduct a quantitative user evaluation of a programming language with a type system that is unfamiliar to most available participants.
2. Showing that Solidity participants in a lab setting frequently insert bugs that lose assets in a small programming task that is representative of a real smart contract programming problem.
3. Showing that most of the Obsidian participants in our study were able to use ownership to solve a security problem, with which participants had great difficulty in earlier versions of Obsidian.
4. Showing that although the participants were able to use ownership effectively, they were less successful in using typestate to obtain static safety guarantees, and further work may be required in this area.

Considering Obsidian's design goal to improve safety with a novel type system, we identified three high-level research questions for our study:

RQ1: Could we obtain actionable data about the usability of a novel programming language

[1] Portions of this chapter previously appeared as [44].

```
asset contract Medicine {}
asset contract Pharmacy
{
  Medicine@Owned med;
  transaction getNewMedicine(Medicine@Owned >> Unowned m)
  {
    med = m;
  }
}
```

What is the error (if one exists) with the getNewMedicine transaction?

- ○ med becomes Unowned, although it should be Owned at the end of the transaction
- ○ m is stated as becoming an Unowned reference, but actually stays Owned
- ○ The owning reference to m is lost
- ● The owning reference to the original Medicine object (med) is lost
- ○ There is no error

Figure 11.1: An example practice question on assets from the Obsidian tutorial (showing the correct answer).

(that uses a type system that would be unfamiliar to our participants) in a short-duration user study (less than one day), which would be representative of real-world smart contract development?

RQ2: Do programmers using Solidity insert more of the kinds of bugs that Obsidian is designed to catch?

RQ3: Strong type systems can impose a usability burden on programmers because the compiler forces programmers to write code so that the compiler can verify various safety properties. Can programmers who were previously familiar with object-oriented programming (but not with Obsidian, typestate, ownership, or linear type systems in general) successfully use Obsidian to complete relevant smart contract programming tasks? If so, is there a significant impact on task completion times?

In RQ1, *actionable data* refers to data that inform programming language designers about their designs in ways that have implications on language design or language training. We used these three research questions to develop task criteria (§ 11.6). Then, we developed three tasks according to the criteria: Auction (§ 11.7), Prescription (§ 11.8), and Casino (§ 11.9). In each task, we refined these high-level research questions into task-relevant ones.

In summary, after the training period (about 90 minutes), seven of the ten Obsidian participants were able to successfully use Obsidian to finish implementing the required small program in our Auction task. In contrast, only two of the ten Solidity participants finished the task correctly. In the Prescription task, seven of the ten Obsidian participants were able to fix a security vulnerability in the program, whereas only two of the ten Solidity participants were able to do so. Six of the Obsidian participants used ownership to do the task, suggesting that ownership is teachable in a relatively short training period.

Five of the Obsidian participants had enough time remaining to do the Casino task (meaning

that the *other* participants felt that their solutions were incomplete when their four-hour time expired). Although four of these wrote solutions that compiled, all four abused the `disown` operator, resulting in asset loss. Among the eight Solidity participants who had enough time for this task, half inserted bugs that resulted in improper asset fabrication or loss, showing that it is still important to pursue approaches to prevent asset abuse. Only one participant, who was in the Solidity condition, finished the task correctly.

After the tasks had ended, we gave participants a post-study survey asking for their opinions about the language they had used. We found that Obsidian participants felt that ownership was useful. They also observed that the tutorial and exercises were effective tools for learning the language.

We come to three conclusions in this chapter. First, the methodology we used, in which we include practice-based training in a traditional quantitative study, is effective for evaluating a novel programming language. Second, despite the strong, unfamiliar restrictions it imposes, Obsidian's ownership and permissions system can be taught to some kinds of programmers in a short period of time in a way that results in many of them being able to use it effectively. That is, for some tasks, Obsidian enables people to be more successful at writing code that is free of the serious bugs that Obsidian detects. Third, without using techniques to detect or prevent asset-loss bugs, when doing some kinds of smart contract programming tasks, programmers will accidentally insert these kinds of bugs.

11.2 The Solidity Language

Solidity [73], which we used for the control condition of this study, is a class-based object-oriented language like Obsidian. Solidity targets the Ethereum blockchain platform [72]. The `function` keyword denotes methods, though methods have transactional semantics. It has no built-in notion of states, but programs can declare `enum`s and use them to represent states. Solidity supports a built-in cryptocurrency called *ether*. Functions that are annotated `payable` can receive quantities of ether; the ether is conceptually sent with the invocation, and the amount is stored in the variable `msg.value`. Each contract instance can own a quantity of ether; this quantity is automatically updated by the runtime when a function receives a payment. Every contract instance is stored on the blockchain at a particular address. The language has a built-in type called `address` to represent these addresses.

Restricting only functions annotated `payable` to receive ether may prevent some kinds of lost-ether bugs, in which clients accidentally send ether to the wrong function. However, this approach does not apply to other kinds of resources, and only covers *whether* the function can receive ether, not whether the function body accounts for it correctly. Likewise, although Solidity includes support for permission modifiers, such as `private`, to protect sensitive functions from being called externally, these modifiers do not prevent the bodies of methods from abusing assets or from being invoked when the receiving contract is in an inappropriate state.

Programs typically implement their own fine-grained accounting mechanism. For example, the `pendingReturns` structure records how much money is owed to each of a number of addresses. Without this mapping, although the contract would still record how much ether it held, the implementation would not be able to track for whom it is being kept.

The `pendingReturns` mapping supports the *withdrawal pattern* [74], which is an Ethereum coding convention that protects against re-entrancy attacks. The possibility of attacks arises because sending ether to a contract can cause the recipient to execute arbitrary code: the recipient can invoke a function on the sender, to which there is already an invocation on the stack. This is dangerous if the funds were sent while the sender was in an inconsistent state. Instead, it is recommended that contracts merely record that they owe ether to the intended recipient and provide a `withdraw` function that recipients can call to retrieve their money. The withdrawal pattern is not used in Obsidian, since Obsidian targets Hyperledger Fabric, which does not have this vulnerability. Because this was a summative study of the programming environments that users would encounter when using each language, we expected participants to use the withdrawal pattern with Solidity and not with Obsidian. This introduced the risk that the use of the withdrawal pattern would cause additional complexity relative to Obsidian.

In Solidity, using the withdrawal pattern typically results in the programmer writing code with arithmetic operations to update balances. In contrast, an Obsidian implementation of the withdrawal pattern would be expected to use linear assets, which the compiler could check for abuse — making the withdrawal pattern safer on Obsidian than Solidity. Although relevant to this study, if Obsidian were used with Ethereum, we expect Obsidian's asset-based approach would guard against bugs in using the withdrawal pattern, regardless whether the opportunity for it arises from the platform design or from the particular API being used.

As an example to illustrate how Solidity compares with Obsidian, Figure 11.2, which is taken from the first task of this study, shows part of an `Auction` contract; the left column shows Obsidian, and the right column shows Solidity.

11.3 Study Design

Our experimental protocol was approved by our university IRB. The experiment began with obtaining informed consent. Participants (§ 11.4), whom we recruited from the university community, were randomly assigned to use either Obsidian or Solidity and given a tutorial (§ 11.5) on their assigned language, during which an experimenter answered any questions they had. Then, the study proceeded with three tasks: Auction (§ 11.7), Prescription (§ 11.8), and Casino (§ 11.9). After the tasks had ended, we gave participants a post-study survey (§ 11.10) asking for their opinions about the language they had used. We compensated participants with a $75 Amazon gift card.

Our participants had a variety of levels of programming skill and were new at programming in Obsidian and mostly new at programming in Solidity. To make it more likely that they would complete at least some tasks and to leverage the skills that participants would acquire by doing the tasks, we gave the tasks to the participants in order of increasing difficulty (according to our experience with pilot studies). All participants worked on the tasks in the same order.

Prior empirical work in programming language evaluation found that testing and debugging programs in the context of empirical studies substantially increases variance. For example, as we discussed in chapter 5, we found that different participants have different levels of thoroughness in writing tests and different levels of debugging skill. When allowed to test and debug, participants frequently spend large amounts of time debugging issues that are not relevant to the experiment.

```
1  main asset contract Auction {
2    Participant@Unowned seller;
3
4    state Open;
5    state BidsMade {
6      // the bidder who made the highest bid so far
7      Participant@Unowned maxBidder;
8      Money@Owned maxBid;
9    }
10   state Closed;
11
12   ...
13
14
15
16
17   transaction bid(Auction@Shared this,
18                   Money@Owned >> Unowned money,
19                   Participant@Unowned bidder) {
20       if (this in Open) {
21         // Initialize destination state,
22         // and then transition to it.
23         BidsMade::maxBidder = bidder;
24         BidsMade::maxBid = money;
25         ->BidsMade;
26       }
27       else {
28         if (this in BidsMade) {
29           //if the new bid  > current Bid
30           if (money.getAmt() > maxBid.getAmt()) {
31             //1. TODO: fill this in.
32             //Can call other transactions as needed.
33             maxBidder.receivePayment(maxBid);
34             maxBidder = bidder;
35             maxBid = money;
36           }
37           else {
38             //2. TODO: return money to the bidder,
39             //   since the new bid was too low.
40             //Can call other transactions as needed.
41             bidder.receivePayment(money);
42           }
43         }
44         else {
45           revert("Can't bid on closed auctions.");
46         }
47       }
48     }
49 }
```

Obsidian condition.

```
contract Auction {
  // the bidder who made the highest bid so far
  address maxBidder;
  uint maxBidAmount;

  // 'payable' indicates we can transfer money
  // to this address
  address payable seller;

  // Allow withdrawing previous money
  // for bids that were outbid
  mapping(address => uint) pendingReturns;

  enum State { Open, BidsMade, Closed }
  State state;
  ...
  function bid() public payable {


    if (state == State.Open) {
      maxBidder = msg.sender;
      maxBidAmount = msg.value;
      state = State.BidsMade;
    }


    else {
      if (state == State.BidsMade) {
        //if the new bid  > current Bid
        if (msg.value > maxBidAmount) {
          //1. TODO: fill this in.
          //Can call other functions as needed.
          pendingReturns[maxBidder] += maxBidAmount;
          maxBidder = msg.sender;
          maxBidAmount = msg.value;
        }
        else {
          //2. TODO: return money to the bidder,
          //   since the new bid was too low.
          //Can call other functions as needed.
          pendingReturns[msg.sender] += msg.value;
        }
      }
      else {
        revert("Can't bid on closed auctions.");
      }
    }
  }
}
```

Solidity condition.

Figure 11.2: A side-by-side example comparing Obsidian code to Solidity code, taken from the two conditions of the Auction task (with minor changes to fit on this page). Code highlighted in yellow represents a correct solution; the rest was given to participants as starter code. In the Obsidian code, line 33 transfers ownership of the object referenced by `maxBid` to the `receivePayment` parameter. The new type of `maxBid` from then until line 35 is `Money@Unowned`. Line 35 re-establishes ownership in `maxBid` by transferring ownership from `money` to `maxBid`.

To focus our participants' time on work related to our research questions, we allowed them to edit their code until they were satisfied, but did not give them an opportunity to test their code. Then, instead of assessing programs on the basis of tests, we inspected their code. We looked both for specific bugs that corresponded to the research questions we were interested in as well as for unrelated bugs. To do this, we developed a rubric for each task, which listed particular bugs to look for; we iterated on this rubric to add bugs that were present in the programs. This approach was made feasible by the relatively small codebases (see Table 11.11). Although it is possible that we missed particular bugs in the code, the same would be true even if we provided unit tests. However, by evaluating by inspection rather than unit tests, we were able to assess code with significant functional problems, similar to how one might grade a student exam by using a rubric rather than by executing unit tests. By using a rubric, partial credit can be awarded and specific mistakes can be identified even when the solution contains significant flaws.

Our decision to allow compilation but not testing was also motivated by our desire to evaluate the ability of Obsidian's type system to detect bugs that might otherwise be introduced. In many cases, it is cheaper to find bugs earlier in the software development process than later, so we focused on whether the kinds of bugs that Obsidian can detect at compile time are ones that are introduced at all and how successfully Obsidian programmers can complete tasks. In Delmolino et al. [59], the programmers had access to a blockchain and still finished their work with code that lost assets; nonetheless, this does reflect a limitation in the study design. Because Obsidian's design is focused on identifying more bugs at *compile time*, allowing compilation but not testing allowed us to focus on our research questions about the usability and effectiveness of the *type system* while using our participants' time as effectively as possible. However, this may have introduced a kind of bias, since the Solidity participants might have found some of their bugs through testing if that had been permitted.

Another approach that could improve external validity is adding code review to the process. We did not include an explicit code review step in which peers inspected the code; although this might more-realistically represent some settings, studies have found that code reviews only find bugs infrequently [53]. Also, adding code reviews to the process would have substantially increased time requirements as well as variance, since the quality of the feedback would necessarily have varied.

11.4 Participants

We recruited participants to our four-hour study with posters at our university, with emails to lists of appropriate degree programs (such as the Master of Software Engineering program), and by advertising to students in relevant courses (e.g., a software engineering course).

Because we advertised the study broadly (the posters were visible to anyone on campus), and because the study assumed that participants already knew an object-oriented programming language, we wanted to make sure that all participants had appropriate preparation. Therefore, we pre-screened participants with an online survey that asked them basic Java questions; we invited respondents who answered five of six questions correctly to participate in the study. The complete pre-screening instrument is in the replication package [42]. The six questions concerned: Java constructor syntax; the definition of encapsulation; whether changes to a list through a reference

	Solidity ($N = 10$)	**Obsidian** ($N = 10$)
Median programming experience, years (range)	9.2 (3.3 - 13)	5.0 (2.3 - 8.5)
Median professional experience, years (range)	1 (0.5-9.0)	0.92 (0.0 - 5.0)
Median Java experience, years (range)	2.0 (0.67 - 4.2)	1.5 (0.25 - 6.0)

Table 11.1: Participant experience (self-reported).

would be visible through another reference; whether methods in interfaces may include bodies; whether abstract classes may be instantiated; and whether concrete subclasses of abstract classes must implement methods that were abstract in the superclass.

Table 11.1 summarizes the previous experience of the experiment participants in each condition. We excluded an additional participant who took so long on the training phase that not enough time was available for more than one programming task[2]. This left 20 participants; 14 of them identified as male, and six as female. Only two participants, both in the Obsidian condition, had no professional experience. Blockchains are targeted at enabling general software engineers in a wide variety of industries to build applications in contexts that lack mutual trust. Since our participants had substantial programming experience and 18 of them had some professional experience, we argue that our participants were representative of at least some kinds of programmers in industry who might be interested in writing smart contracts.

11.5 Training

We provided a web-based tutorial (implemented with Qualtrics, and included in the replication package), which stepped participants through web-based documentation and exercises for their assigned language. Some of the exercises were to be completed in Visual Studio Code, which was configured with a compiler. Some of the questions were multiple-choice; for these, the tool automatically showed participants if they had entered an incorrect answer. An experimenter was available to answer questions. Participants were told that they should try to get all of their questions addressed during the training phase, since no questions could be answered after training was completed.

We included practice questions in the tutorial to ensure that participants absorbed the material (we had found that without practice questions, participants skimmed the material without mastering the concepts; see § 11.5). Figure 11.1 shows an example practice question. Figure 11.3 shows a sample from the Obsidian documentation.

To make the two experimental conditions as similar as possible, even though Solidity includes no support for ownership, typestate, or assets, Solidity participants received a tutorial that explained these concepts and recommended using comment-based annotations. If participants asked about the utility of these annotations, we argued that this was similar to how one might write preconditions or postconditions in comments. Table 11.2 summarizes the distribution of

[2]That participant spent 3 hours and 11 minutes on the tutorial, which was three standard deviations above the mean.

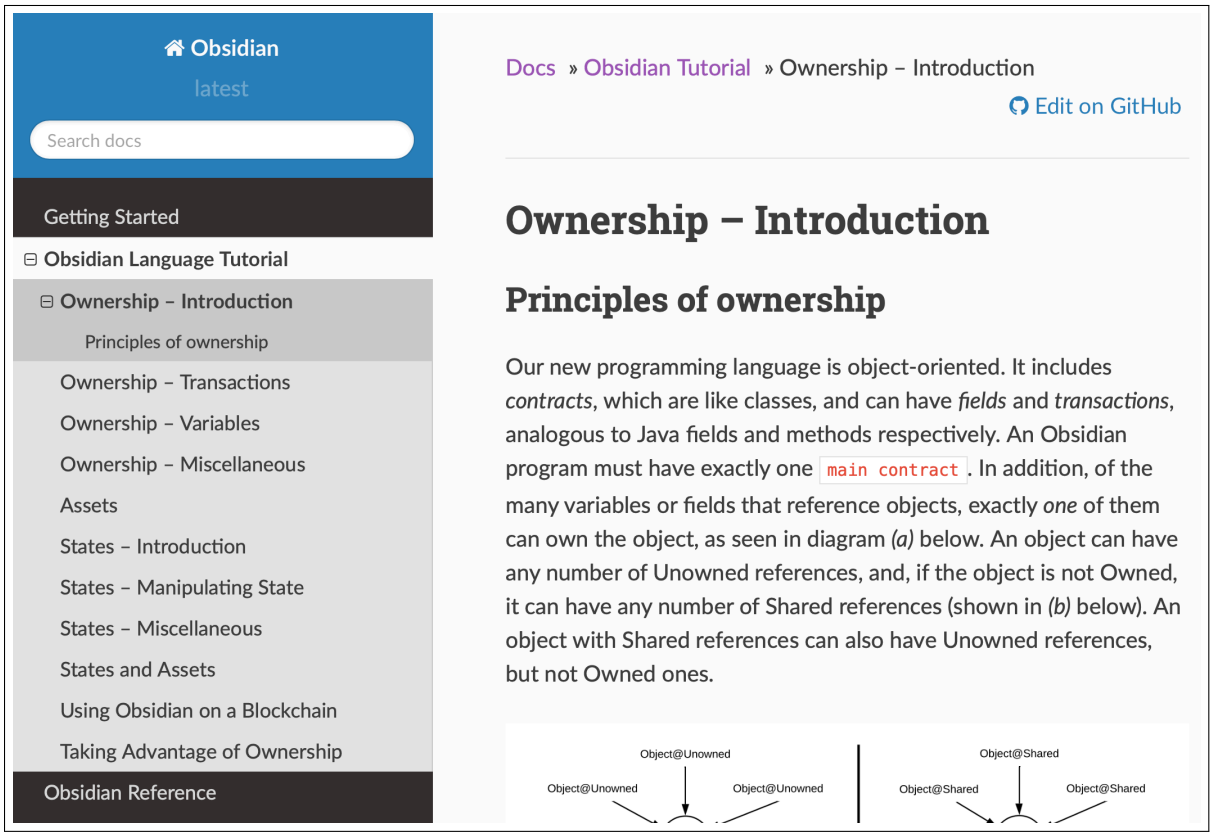

Figure 11.3: An example from the web-based Obsidian tutorial.

	Solidity ($N = 10$)	**Obsidian** ($N = 10$)
Average (standard deviation)	86 (28) min.	98 (31) min.
Range	39 to 138 min.	50 to 148 min.

Table 11.2: Training times in Solidity and Obsidian conditions.

times participants spent on the tutorial in the two conditions. If we had trained only the Obsidian participants in ownership, for example, any differences in behavior could have been attributed to the training and not to the language they were using.

Solidity participants received training on the *withdrawal pattern* and its necessity for security reasons, as well as language-specific details such as the `payable` keyword and the concept of *ether*.

11.6 Task Selection

Based on our research questions, we sought a collection of tasks for our study that met several criteria. We selected criteria, shown in Table 11.3, to help us identify tasks that would address our research questions. Because these criteria were similar to those we used in developing the earlier

Name	Criterion	RQs	Tasks
C1	All tasks should reflect smart contract use cases that have received some attention in the blockchain community.	RQ1	Auction, Prescription, Casino
C2	At least one open-ended task that could be used to evaluate whether Obsidian participants could create their own typestate-oriented interfaces (rather than only implementing according to a well-defined specification).	RQ3	Casino
C3	At least one task should put Solidity participants in a position where they might accidentally lose an asset. For Obsidian participants, the corresponding question is whether they are able to complete the task in spite of the strictures of ownership.	RQ2, RQ3	Auction, Casino
C4	In the previous qualitative study of an early version of Obsidian, participants found it very challenging to use ownership (§ 8.3). Therefore, at least one task should assess whether the changes to Obsidian since its original design have addressed the difficulty of using ownership to restrict the use of assets.	RQ3	Prescription, Casino

Table 11.3: Alignment between task design criteria and the tasks we designed.

studies, we were able to adapt the previous tasks for this purpose.

The tasks and their results are described in detail in the following sections. Complete task materials are available [42]. Although we told participants that we might interrupt them eventually if they needed to move on to the next task, we reduced time pressure by not telling them specific per-task time limits.

11.7 Auction Task

In the *Auction* task, we asked participants to fill in missing code in an implementation of an English auction, in which bids are made openly and the highest bidder wins. To increase external validity (criterion C1 in Table 11.3), we modeled the task after an example from a Solidity tutorial [76]. We required that all bids be accompanied by funds to ensure that the winning bidder will pay for the item. When a bid is exceeded, the original bidder should receive a refund of their bid. We gave the participants 30 minutes to complete the task.

Figure 11.2 shows the `bid` transaction that was provided to participants as well as a sample solution. In the first subtask, marked by `// 1. TODO`, participants needed to write code to refund the existing bid to the previous bidder, whose address was stored in `maxBidder`, and record the new bid (`money`). In the second subtask (`// 2. TODO`), participants needed to refund the bid to the bidder. The code in yellow shows a correct answer. In both cases, there was an opportunity for *asset loss*: if participants overwrote the old `Money` reference (stored in `maxBid`), then the old bid would be lost. In Obsidian, the compiler would report an error if this happened; in Solidity, there was no protection against that mistake. This opportunity for error

reflected task criterion C3.

We refine the high-level research questions for this task (using numbering that reflects the original research questions in § 11.1):

RQ 2.1: How frequently do Solidity participants accidentally lose assets in the Auction task?

RQ 3.1: Overall, do completion times differ across conditions?

RQ 3.2: Overall, are participants more likely to finish Auction (and do so correctly) if they use Obsidian rather than Solidity?

11.7.1 Auction Results and Discussion

Table 11.4 summarizes the results of the Auction task; errors are shown in Table 11.5. Nine Solidity participants said they were done with the task before the 30 minutes expired; among these nine, the average time was 12 minutes (95% CI: [6.8, 17.4]). Eight Obsidian participants said they were done with the task before running out of time; among these eight, the average time was also 12 minutes (95% CI: [6.4, 18.3]). Thus, for **RQ 3.1**, the difference in times was not significant. Two participants completed the task correctly in the Solidity condition; seven completed the task correctly in the Obsidian condition. The difference in success rates (summarized in the first two rows of Table 11.4) is statistically significant, with $p \approx .015$ (Fisher's exact test; odds ratio 0.053). We conclude for **RQ 3.2** that participants who finished were more likely to finish correctly if they used Obsidian than if they used Solidity.

Of the two Obsidian participants who did *not* finish the Auction task in time, one (P45) was confused about the semantics of the `::` field initialization operator, attempting to use `BidsMade::maxBid` to refer to the current value of the `maxBid` field rather than the future value after a state transition. This misconception led to a compiler error message that the participant did not find helpful. The other participant also received a confusing error message: although the code invoked a transaction that did not exist, the error message pertained to ownership of the transaction's parameter. A more mature compiler with better error messages might have helped the participants finish the task.

In **subtask 1** (starting at line 31 in Figure 11.2), participants needed to record the new bid and refund the old bid. We found the following errors among the Solidity participants who said they were done:

1. Loss of previous refunds: the correct implementation *added* the new refund to any prior refund. Four participants used = instead of +=, overwriting any old refund (in line 33).

	Solidity	**Obsidian**
Completed task correctly	2	7
Completed task with bugs	7	1
Time in min., completed tasks only; 95% CI	12; [6.8, 17.4]	12; [6.4, 18.3]
Did not complete the task	1	2

Table 11.4: Auction task results. N=10 in each condition.

	Solidity	Obsidian
Ran out of time	1	2
Lost an asset in either subtask	7	0
Subtask 1		
omitted refund of old bid	3	0
overwrote old refund	4	0
refunded to wrong bidder	0	1
Subtask 2		
overwrote old refund	4	0
refunded via `transfer()` instead of `pendingReturns`	4	N/A

Table 11.5: Errors in Auction task. N=10 in each condition.

2. Omission of refund: three participants neglected to refund the previous bid (e.g., omitting line 33).

All eight of the Obsidian participants who said they were done did so without losing any assets, since otherwise the compiler would have given an error. However, one participant refunded the old money to the *new* bidder instead of to the *previous* bidder.

While doing the task, two of the Obsidian participants received a compiler error indicating that they had lost an asset. For example:

```
auction.obs 37.28:  Variable 'maxBid' is an owning reference
to an asset, so it cannot be overwritten.
```

Both of these participants successfully fixed the error.

In **subtask 2** (starting at line 41 of Figure 11.2), participants needed to refund the new bid, since it was not larger than the previous bid. Among the nine Solidity participants who said they finished the task, two refunded the bid properly (using `pendingReturns`). Four refunded via `transfer`, which would not have resulted in asset loss but was inconsistent with the documentation we gave them. The documentation specified to use the withdrawal pattern to be consistent with the typical recommendation when using Solidity. Four attempted to refund via `pendingReturns` but, as in the first subtask, overwrote any previous refund, potentially losing money.

One might argue that the potential for asset loss due to improper use of `pendingReturns` was due to the need to use the *withdrawal pattern* [74], as discussed above. However, the particular bug we observed was due to participants overwriting an integer rather than adding to it, and we infer that arithmetic errors are likely common when manipulating assets manually. Obsidian protects against these bugs by encouraging programmers to design APIs that use assets to represent money rather than raw integers.

Of the seven Obsidian participants who completed the Auction task successfully, while they were working, two received compiler errors indicating that they had lost assets. One additional participant, P45, got an error about a lost asset, but did not finish the task because they were confused about the use of the `::` operator.

We conclude (**RQ 2.1**) that asset loss was frequent among Solidity users, and *more* frequent than among Obsidian users, who did not lose any assets ($p \approx .002$, Fisher's exact test). This difference may have been caused by a combination of differences in language design and differences in API design, but the different API designs were related to different language design choices.

11.8 Prescription Task

To address task criterion C4, we gave participants a short `Pharmacy` contract (43 lines in Obsidian or 46 lines in Solidity including whitespace). The code included an example to show how the contract was vulnerable to attack. Although a `Prescription` was specified to only permit a fixed number of refills, a `Patient` could invoke `depositPrescription` on more than one `Pharmacy` object, resulting in the patient being able to refill the prescription the given number of times at *each* pharmacy. We asked participants to fix the bug, avoiding runtime checks if possible. In Solidity, for example, `depositPrescription` had the signature below:

```
function deposit(Prescription p) public returns (int);
```

In Obsidian, the starter code provided this signature:

```
transaction deposit(Prescription@Shared p) returns int;
```

In Obsidian, it sufficed for participants to change the signature so that the `Pharmacy` acquired ownership of the prescription object:

```
transaction deposit(Prescription@Owned >> Unowned p) returns int;
```

In Solidity, in contrast, since there is no static feature that would make the above safe, participants had to implement a global tracking mechanism across all `Pharmacy` objects.

The task was based on a task from the usability study, which found that users of an earlier version of Obsidian had great difficulty using ownership to fix this problem (§ 8.3). For example, some participants in that study thought about ownership in a dynamic way (for example, writing `if` statements to test ownership) or were confused about when ownership was transferred between references. The version of the language used in the present study includes changes that resulted from that work, such as fusing typestate and ownership in the language syntax, making ownership transfer explicit in transaction signatures, and removing local variable ownership annotations. Our research questions were centered around evaluating the revised language:

RQ 3.3: What fraction of Obsidian participants could use ownership to fix the multiple-deposit vulnerability?

RQ 3.4: Does using ownership to prevent the multiple-deposit vulnerability take less time than using a traditional dynamic approach?

We gave participants 35 minutes to complete the task. Because ownership-based approaches are not checked by the Solidity compiler, we wanted to allow Solidity participants who proposed ownership approaches to try again. Therefore, when participants informed the experimenter that they were done, the experimenter inspected their code. If they had used static notions of ownership instead of a dynamic approach, participants were permitted to try again in the rest of their 35 minutes.

	Solidity	Obsidian
Attempted a static solution	5 participants	6 participants
Correct static solution	N/A	6
Attempted a dynamic solution	6	3
Correct dynamic solution	2	1
Made `Prescription` mutable	2	1
Completed within time limit	3	9
Mean time among successful participants; [95% CI]	20 min.; [0, 45]	22 min. [12, 33]
Mean time among successful participants after removing Solidity time spent on static attempts	18 min.	22 min.

Table 11.6: Summary of Prescription task results. Times are shown as mean (standard deviation). N=10 in each condition. Two Solidity participants tried both static and dynamic approaches, and one Solidity participant made no changes, resulting in 11 Solidity attempts.

11.8.1 Prescription Results

Table 11.6 summarizes the results. Regarding **RQ 3.3**, six of the ten Obsidian participants successfully used ownership to solve the problem. The dynamic Obsidian solution that we judged to be correct tracked global state by making `Prescription` mutable, despite a comment indicating that `Prescription` should be immutable.

Five of the ten Solidity participants tried to use ownership, even though Solidity does not check ownership. Only three of the Solidity participants said that they were done within the time limit, and of those, only two had a correct solution. The incorrect Solidity solution attempted to solve the problem by making `Prescription` mutable to track remaining refills globally, but in addition, although the participant tried to track the number of refills across all pharmacies, the code did not update the global number of refills when refilling a prescription.

We separated time Solidity participants spent on static solutions from the time spent on dynamic solutions, since these attempts were likely prompted by the training materials; Table 11.6 shows both sets of times. However, we included all time spent by Obsidian participants on correct solutions because it is realistic that some Obsidian users would have tried each approach.

Regarding **RQ 3.4**, we did not observe a significant difference in completion times. However, a large fraction of Solidity participants spent significant portions of their time attempting static solutions. In retrospect, it might have been more informative to have told Solidity participants explicitly that they needed to use a dynamic solution, and this approach might have led to a more interesting comparison. However, due to the small amount of code required for the static solution, we suspect that there is a significant learning effect to be leveraged here in Obsidian, and the participants who succeeded could likely do so again in a similar situation much faster. Furthermore, the fact that only two of three Solidity participants who finished this task did so correctly underscores the benefit of static enforcement of these kinds of safety properties.

11.8.2 Prescription Discussion

One might have expected that applying ownership would be challenging, since the concept was new to the participants, but since only half of those who said they had completed a dynamic solution had correct solutions, it would appear that using the new static construct may not be harder than writing global state tracking code. In fact, using ownership to solve programming problems is *teachable*: six of nine Obsidian participants who completed the task used ownership to do so. We expect that the remaining three could be taught to do so with additional practice.

In the earlier study (§ 8.3), which used a very similar task, four of six participants were able to solve the problem, but all of them received significant help from the experimenter. The remaining two did not solve the problem even with help. Although those numbers cannot be compared with the results of this study directly, the fact that six of our participants were able to complete the task without any help suggests that the changes since then have improved usability.

Security experts have long argued in favor of immutable data structures [165, 180], which is one reason why we specified that `Prescription` was immutable. However, these results point out that this approach may not be tenable: specifications of immutability may be ignored or removed, and attempts to maintain immutability require substantial work, which may itself be bug-prone. Indeed, when language-based mechanisms do not provide the required safety properties (chapter 4), it may be safer and cheaper to use a *mutable* design than to bear the cost of immutability. In this case, making `Prescription` mutable would have obviated the need to implement separate data structures keeping track of how many refills each prescription had left, since that information could be kept in the `Prescription` object directly. That approach appeared to be more natural for the participants, and certainly required less code.

The benefits of Obsidian's ownership system in the Prescription task contrast with the benefits of other kinds of ownership. For example, ownership in Rust [164], which is understood to be one of the more challenging aspects of learning Rust [220], introduces constraints on mutation. However, in Rust, only owning references can mutate objects. In Obsidian, mutation is restricted only as much as is needed to provide sound typestate specifications, since concurrency is not a concern. Therefore, in Obsidian, only changing *which named state an object is in* is restricted, and then only through `Unowned` references. This contrasts with Rust, in which *all* modifications to fields of objects are restricted. Six of 10 participants were able to use the linear aspects of ownership alone in the Prescription task, suggesting that languages that adopt just linearity (and not mutability restrictions) may be usable. Perhaps by integrating a more flexible permissions system, languages such as Rust could be made more convenient for common cases, and thus have a more gradual learning curve. Alternatively, making these changes in Rust might come at the expense of expert efficacy, so the impact of these changes in the long term would need to be evaluated.

Although six of the Obsidian participants used ownership successfully, four participants did not. One of the four participants did not complete any of the three tasks. Another "fixed" the issue by modifying code in `Patient` that we had provided as an example of how a nefarious patient might exploit the bug; perhaps this participant did not really understand the task. The other two seemed to need more time studying ownership in order to use it effectively.

The instructions included: "Please use what you have learned today to fix this problem (avoiding runtime checks if possible)." Perhaps as a result, a similar fraction of participants tried

to use ownership in both conditions. We wanted to encourage the Obsidian participants to use ownership so that we could assess to what extent they could use it effectively, but the instructions persuaded some of the Solidity participants to use it even though doing so was not checked by the compiler.

11.9 Casino Task

The casino task was more open-ended than the other tasks, addressing criterion C2. We gave participants a web page with a diagram showing invocations that needed to be supported (Figure 11.4). The web page included a list of requirements:

1. If a `Bettor` predicts the outcome correctly, the `Bettor` gets twice the `Money` they put down. For example, if `Bettor` b puts down 5 tokens on the correct outcome, they should receive 10 tokens after the Game is played.
2. If the `Bettor` predicted incorrectly, the `Casino` keeps their tokens.
3. Bets can only be made before the `Game` starts.
4. Winnings can only be distributed after the `Game` is finished.
5. `Bettor`s must collect winnings themselves from the `Casino` after a Game by calling code, which you need to write. Until winnings are collected, the `Casino` keeps track of them.
6. A `Bettor` can have one active bet per game. If a `Bettor` bets more than once, their original bet should be replaced by the new one and any previous bet should be refunded.
7. A `Bettor` MUST put down tokens at the same time that they're making a Bet.
8. If the `Casino` does not have enough tokens available to pay out winnings, the invocation to collect winnings can fail.

We provided starter code for `Casino`, `Game`, and `Bet`. Obsidian participants also received implementations of appropriate containers (Solidity has suitable built-in containers).

We used the Casino task to investigate five research questions:

RQ 2.2: How frequently do Solidity participants lose assets in the Casino task?

RQ 3.5: To what extent do Obsidian participants leverage typestate in transaction signatures to avoid dynamic checks (criterion C2)?

RQ 3.6: In both versions, programs represented funds with `Token` objects. Does Obsidian's type system help participants avoid losing `Token` objects compared with Solidity (criterion C3)?

RQ 3.7: Do Obsidian participants view `Token` objects as resources that should not be created or destroyed, or as data, which could be created and destroyed as needed (criterion C4)?

RQ 3.8: How do task completion times compare between Solidity and Obsidian participants (criterion C4)?

Of the original ten participants in each condition, we excluded one Obsidian participant who should have received an error from the compiler but, due to a bug, did not. We also excluded one Solidity participant and four Obsidian participants who did not have enough of their four hours

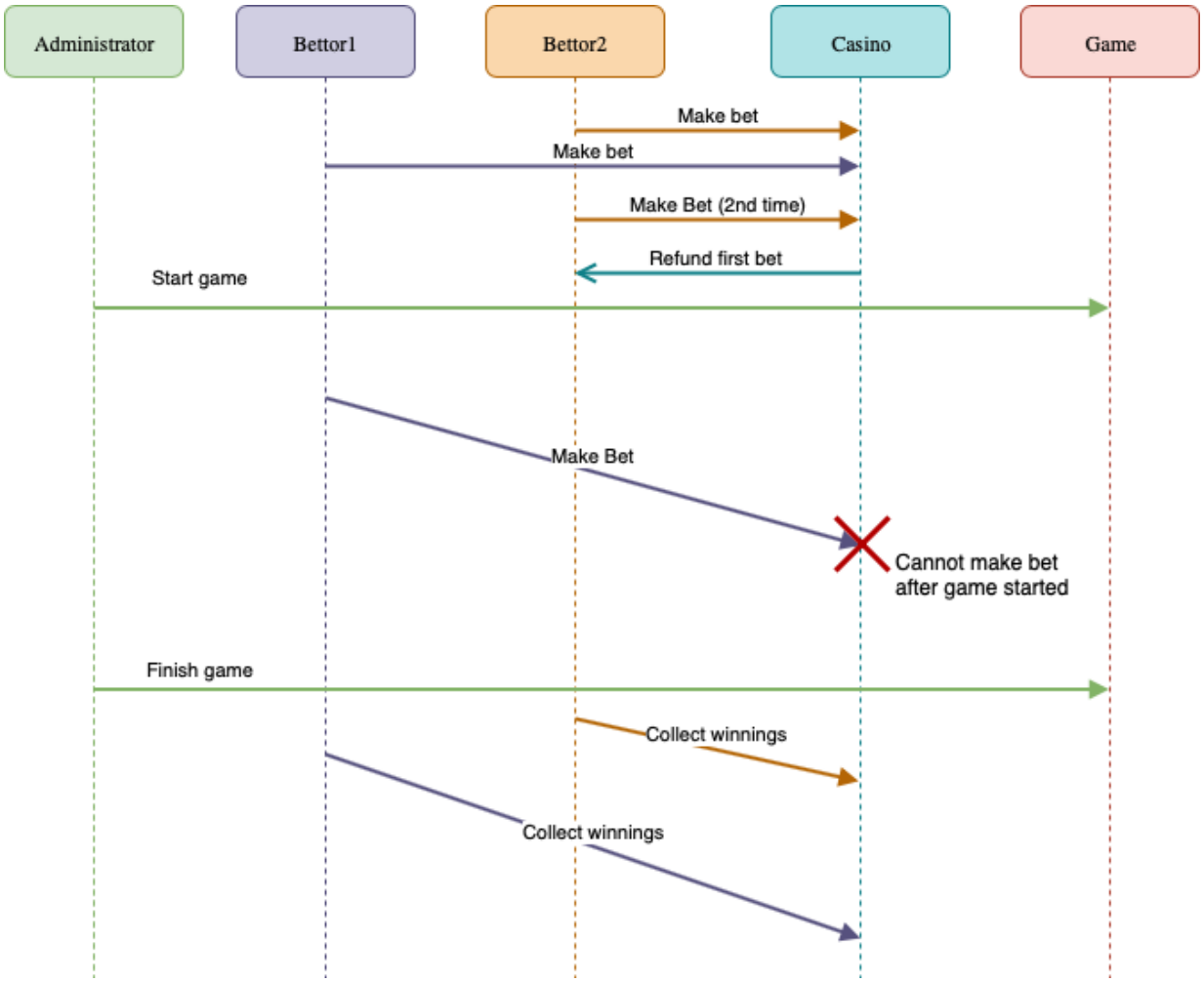

Figure 11.4: Sequence diagram given to participants to show what operations the Casino contract should support.

remaining, and in the time available, were not satisfied with their solution. This left nine Solidity participants and five Obsidian participants whose results we analyzed. The results below pertain to these 14 participants. The discrepancy between the number of participants who had enough time across the two conditions was due to two factors: the faster task completion times by Solidity participants, and the high variance among Obsidian participants for time required for tasks that occurred *before* Casino.

Of the 14 participants whose results we analyzed, one participant in the Obsidian condition gave up after 1 hour, 15 minutes. That participant had chosen an unnecessarily difficult implementation strategy, requiring implementing a new container (implemented as a linked list). Also, the participant delayed trying to compile until after writing a lot of code, resulting in a large collection of compiler errors. The remaining four Obsidian participants all wrote code that compiled successfully. One participant in the Solidity condition gave up after 39 minutes, having received a parser error that they were not sure how to fix.

The Casino task results should be interpreted in the context of some additional limitations relative to the other tasks. The results of a comparison between conditions on this task may be biased because the Obsidian participants who were included for analysis in this task are those who completed the earlier tasks fastest. As a result, the Obsidian participants may have been stronger programmers on average than those in the Solidity condition (in which almost all the participants had time to try the task).

Casino was more open-ended than the other tasks, resulting in more variance, as participants made varying implementation choices. Given this, the small numbers, and the potential bias, we use this primarily as an opportunity to develop hypotheses, design insights, and identify future opportunities for improvement.

11.9.1 Casino Results and Discussion

We analyzed the code participants wrote, comparing the code to the requirements we gave them. As with the other tasks, in order to isolate the bugs from each other, our analysis was manual; the source code produced by each participant is included in the replication package [42]. Results are summarized in Tables 11.7 and 11.8. Before discussing the results for the four research questions, we discuss the errors that participants made while working on Casino.

Except for the one Solidity participant who completed the task correctly, all of the other participants across both conditions inserted various bugs. For example, three participants in each condition failed to emit an error when attempting to collect winnings from the Casino when the Casino lacked enough resources to distribute them.

The most common bug among Solidity participants was some kind of token loss or improper fabrication; four of the eight made this mistake, addressing **RQ 2.2**. For example, one participant, when accepting a new bet, first credited the bettor's account for any prior bet, and then returned if the bet amount exceeded the bettor's balance. This meant that a second bet, when disallowed, would incorrectly fabricate tokens out of thin air, since the code that debited the bettor's balance occurred after the early `return`. Another participant neglected to debit accounts for extra wagers, also fabricating tokens out of thin air. This shows that protection against incorrect asset manipulation *is* important.

	Solidity	Obsidian
Had enough time to try Casino	9	5
Completed Casino with a program that compiled	8	4

Table 11.7: Summary of Casino task completions.

	Solidity ($N = 8$)	**Obsidian** ($N = 4$)
Completed task correctly (no identified bugs)	12.5%	0%
Winnings collection emits error if Casino is out of tokens	62.5%	25%
Casino keeps tokens when a bet is lost	100%	50%
Bettor's extra bets result in refunds	100%	75%
Only used disown safely	N/A	0%
Managed tokens correctly (not fabricating or losing them)	50%	0%
Mean completion time	37 min.	64 min.

Table 11.8: Summary of Casino task results among completed programs that compiled, showing correct solution rates among errors made by more than one participant.

All four Obsidian participants who finished the Casino task used `disown` improperly to throw away assets. We found this surprising, since we had warned the participants against improper use of `disown`. The tutorial included an example of how `disown` might be needed inside the implementation of a `Money` contract, and wrote below the example:

> IMPORTANT: `disown` should be used only when you really want to throw something out. Above, `disown` is required because of the manual arithmetic used to manipulate `amount` inside the implementation of `Money`, but it is not needed in most normal code.

It would not have sufficed to remove `disown` from the language; in addition to the fact that it is needed in certain (rare) cases, programmers could build a `Trash` contract to hold discarded objects, thus suppressing any errors the compiler would emit. We have several hypotheses regarding why `disown` was abused:

- Some participants may have used `disown` to silence the compiler when they felt they had a correct solution. One participant discarded the old wager, using money from the casino's pot to pay out bets. The participant also disowned the bet when a losing bettor tried to retrieve their winnings (a correct solution would have put the tokens in the casino's pot).
- Some participants may not have read or understood the tutorial's warning about `disown`; in retrospect, we should have assessed understanding explicitly.
- Some participants did not sufficiently understand the notion of assets. For example, when disbursing winnings, one solution disowned the previous wager and created new `Tokens` when needed, rather than reusing the tokens from the wager.
- Some participants may have used `disown` as a workaround for an unsolved problem. In

code to accept a new bet, one participant disowned any previous bet by that bettor, and then wrote: `// Currently just throws the bettor's money away and hopes they find it eventually.`

This motivates a question for future research to characterize the use of these *escape hatch* constructs, which can be used in both safe and dangerous ways. Some languages include warnings in the names of such constructs, as in `unsafePerformIO` in Haskell [142], but such approaches may not be effective in explaining the danger.

Risk compensation refers to the idea that people compensate for safety features by taking additional risks. For example, drivers may drive faster when wearing seat belts because the seat belts may mitigate the risk caused by higher speeds. However, in many cases (such as in the seat belt example), overall risk is reduced despite potential compensating behavior [97]. Further study is needed to consider the question of whether risk compensation occurs with strong type systems. Perhaps some participants who used `disown` assumed that if there were a bug, the compiler would report it.

Regarding **RQ 3.5**, the `Game` contract that we provided defined states for `Game` (`BeforePlay`, `Playing`, `FinishedPlaying`). When implementing `Casino`, participants wrote code in their `Casino` transactions (to implement requirements 3 and 4) to dynamically check the state of the `Game`. An alternative approach, which we had expected some participants to use, would have involved observing that the state of `Game` was related to the state of `Casino`. If a `Casino` was in a state that permitted bets, then the `Game` must not have started. Likewise, collecting winnings was only possible from a `Casino` whose `Game` had ended. Our starter code defined states in `Game`. Therefore, participants could have avoided dynamic checks in `Casino` by defining multiple states in `Casino`, each of which corresponded to a particular state of `Game`. All of the participants added the dynamic checks. The participant who gave up tried to check the state with a static assertion, evidently not understanding that the assertion was static, not dynamic.

The lack of usage of static state information is unfortunate because it represents a missed opportunity to rule out bugs in calling code. Perhaps more-experienced programmers would be more interested in leveraging this language feature; alternatively, it might require more training or a more-convenient language design. The design the participants chose may have been best given their incentives; the typestate-based approach would have required adding more structure to reflect the typestate relationships between `Casino` and `Game`. However, motivated by the results of this study, we hope to consider future language design changes that make typestate coupling of different objects convenient.

Perhaps *creating* new interfaces that use novel verification-related features (such as typestate) is harder than *consuming* them, in which case further research should consider novel ways of scaffolding interface design and creation. In Obsidian, it is possible to use states to define different fields (each with its own typestate specification) for each possible state of referenced objects. This approach would couple the states of the two objects, letting the programmer avoid runtime checks that would be otherwise required, but we did not train participants in this approach. Perhaps this technique is sufficiently subtle that we should have provided training directly.

Surprisingly, in **RQ 3.6**, we observe that in fact, Solidity participants were probably more likely to correctly have the casino keep tokens when a bet is lost (Table 11.8, $p \approx 0.09$, Fisher's exact test). Likewise, Solidity participants may be more likely to successfully issue refunds

for bets after the first bet ($p \approx 0.33$, Fisher's exact test). We believe this is related to abuse of `disown` by Obsidian participants.

Regarding **RQ 3.7**, all four of the Obsidian participants who wrote solutions that compiled treated tokens as data rather than assets, i.e., at some point in their code, they either created new tokens or disowned tokens. The Obsidian participant who gave up did not do either of those things, likely viewing tokens more as assets, but became mired in a list of type errors. We conclude that use of assets was likely *not* natural for our participants. This interpretation is consistent with the post-study survey results (§ 11.10), in which we observed that participants said they felt ownership and states were more useful than assets.

For **RQ 3.8**, Obsidian participants spent significantly longer on the Casino task than Solidity participants did ($p \approx 0.02$, Mann-Whitney U test, $d \approx 1.9$). Therefore, the stronger type system provided by Obsidian likely had a significant cost in development time for our participants. We hypothesize that this cost is greater with more open-ended tasks, which would explain why we did not observe this difference in the two prior tasks. Of course, the additional cost may be worthwhile since Obsidian would rule out some classes of bugs statically, particularly when used by skilled programmers who do not abuse `disown`. The time to task completion in this experiment might relate in practice to the time required to complete a *prototype* version of software rather than the time required to create a production-quality version.

11.10 Post-study Survey Results

We conducted a post-study survey asking participants about their opinions. We asked participants (on a 5-point scale) how well they understood particular concepts and how useful they thought those concepts were. The results from participants who completed the survey *before* being debriefed from the study are summarized in Table 11.9. Although the Obsidian participants said they thought ownership was significantly more useful than the Solidity participants did ($p \approx 0.002$, $d \approx 2.5$), the Solidity participants indicated that they felt they understood states better than the Obsidian participants did ($p \approx 0.04$, $d \approx 1.3$). Perhaps the existence of an unfamiliar `state` construct in Obsidian, or the unfamiliarity of the relationship between states and types, led to less confidence. There was no significant difference in views of the utility of states. This may be due to the tasks we gave, which did not particularly rely on the static aspects of states.

We also compared across questions. Obsidian participants said that they felt both ownership and states were *more useful* than assets ($p \approx .0027$, $d \approx 2.1$ and $p \approx .037$, $d \approx 1.2$, respectively, according to a Mann-Whitney U test and Cohen's d). The differences between perceptions of understanding between ownership and assets and between states and assets were not significant at $p < .05$. Of course, perceptions of understand and utility were influenced by the particular tasks the participants did and their perceived success at doing those tasks, but these results correspond with the Obsidian participants' failure to regard tokens as assets in the Casino task.

The survey also included a free-form box in which participants could respond to the question "Please use this space to write any additional comments you have about the language or the study." We reviewed the free-form comments, and describe the most interesting ones here. Due to the sparse and brief nature of the data, we did not conduct a more formal analysis. The complete data are included in the replication package.

Several of the Solidity participants expected that the language would have included constructs for states or ownership. For example, one participant wrote in the post-study survey:

> It also seemed like there should be some syntactic sugar for writing things like:
>
> ```
> enum State { Foo, Bar, Buzz }
> State s
> ```
>
> since they are so common.

Three Solidity participants expected that the compiler would check ownership. For example:

> On [semantics] — I was hoping ownership / assets / states would be statically verified. When I wrote code during the 2nd phase of the study, I found that I didn't really document ownerships / asset status.

Similarly, from another Solidity participant:

> I think it would be nice to have some static analysis to check ownership information rather than relying on the programmer to have good comments documenting ownership because in practice documentation is never perfect and often overlooked.

Thus, the Solidity participants asked for features in Obsidian.

Participants using Obsidian had differing opinions regarding how easy it was to learn. One wrote:

> The tutorial and the exercises are well-written and they helped me a lot in understanding the concepts of new language!

Another Obsidian participant wrote:

> The smaller coding exercises were nice to follow and complete. The open-ended part was a little overwhelming to finish, for somebody that just got introduced to new concepts of ownership, states and assets.

One Obsidian participant commented on how ownership seemed natural after some practice:

> . . . the general concepts of ownership [were] a little unintuitive but after working with the language they started to make more sense and seem more natural. . . .

11.11 Summary of Results

In order to summarize the overall results across the three tasks, we computed the fraction of successfully completed tasks for each participant, taking into consideration the tasks that were not included in the analysis above (only nine Solidity participants and five Obsidian participants in the Casino task). The median fraction of tasks completed by Solidity participants was 0%; the median fraction of tasks completed by Obsidian participants was 67% (since these are ratios, we do not take their mean). We compared these fractions using a Mann-Whitney U test, observing a significant difference with $p < 0.008$. Table 11.10 summarizes *successful* task completions: tasks that were completed correctly within time limits.

We now return to the initial research questions.

> RQ1: Could we obtain actionable data about the usability of a novel programming language (that uses a type system that would be unfamiliar to our participants) in

a short-duration user study (less than one day), which would be representative of real-world smart contract development?

Our results regarding the other research questions show that we were able to identify both strengths and weaknesses of Obsidian relative to Solidity in the context of smart contract programming tasks that were drawn from real-world scenarios. Perhaps more importantly, the study led to insights about the type system and its potential risks that can be used to refine future language designs. For example, strong type systems can be learned and used to significant benefit in short periods of time, but escape hatches can be frequently abused.

RQ2: Do programmers using Solidity insert more of the kinds of bugs that Obsidian is designed to catch?

In Auction, we observed that Solidity participants lost assets frequently (seven of nine Solidity participants who finished lost assets) (RQ 2.1). Similarly, four of eight Solidity participants lost assets in the Casino task (RQ 2.2). We conclude that linear type systems likely have value in helping programmers of smart contracts detect bugs earlier. However, the abuse of `disown` in Casino shows that training or language refinement may be needed to help users of linear type systems obtain these guarantees effectively.

RQ3: Can programmers who were previously familiar with object-oriented programming (but not with Obsidian, typestate, ownership, or linear type systems in general) successfully use Obsidian to complete relevant smart contract programming tasks? If so, is there a significant impact on task completion times?

In Auction, we observed no significant difference in completion times, which is promising for strong type systems (RQ 3.1). Furthermore, participants who used Obsidian were more likely to finish Auction correctly (RQ 3.2). In Prescription, 60% of Obsidian users were able to use ownership to fix the vulnerability we gave them, suggesting that earlier work iterating on the language design using user-centered methods may have been effective in making the language more usable than it had been initially (RQ 3.3). Using ownership does not necessarily take less time than using a dynamic approach, but seven of the ten Obsidian participants successfully completed the Prescription task, suggesting that enabling use of ownership can empower programmers to be more effective (compared to two of ten Solidity participants who succeeded) (RQ 3.4). In Casino, we observed that all of the Obsidian participants who finished the task abused the `disown` keyword, which serves as a caution that escape hatches from safety features are easily misused; this misuse can significantly hamper a language's ability to achieve its safety goals (RQ 3.5). Likely due to this misuse, Solidity participants were more likely than Obsidian participants to have the Casino keep funds from losing bets and to issue refunds correctly for revised bets. Likewise, the Obsidian participants appeared to create and destroy tokens at will, rather than treating them as assets; more training is likely required if we want programmers to achieve the safety benefits of adopting this perspective, which may not be a particularly natural one for them. Finally, Obsidian participants spent significantly longer on the task than the Solidity participants did, confirming that in some more complicated tasks, a stronger type system likely increases the cost of an initial implementation.

There has been long-standing debate about a hypothesis that dynamically-typed languages may be better for prototyping work than statically-typed languages [90, 125]. Our results provide a limited form of support for this hypothesis, since the Solidity participants were able to finish the

Casino task faster than the Obsidian participants. Of course, since there were many differences other than type system between the two languages, the study does not address this hypothesis directly.

11.12 Limitations and threats to validity

The pre-screening instrument may have introduced bias, either by being unrepresentative of common content knowledge among typical object-oriented programmers or by discouraging some people from participating. The student participants may not be representative of the population of smart contract programmers. However, since most of the students had some professional experience, they were likely representative of entry-level programmers in industry [187]. Also, other studies have found no significant differences in code correctness in programming studies between students and non-students [2]. The tasks were more constrained than real-world programming tasks, although smart contracts tend to be small in practice, averaging 322 lines [155]. Table 11.11 describes solution lengths in our study.

The participants were new to the programming languages and to smart contract development in general, so it is possible that experienced programmers would have behaved differently. Although we tried to infer how particular aspects of the languages and their type systems affected participants' behavior and performance, because this was a summative study, the results may have been influenced other aspects of the experience, such as the way in which different aspects of the type systems interacted or the details of the particular tasks we gave participants.

The order of the tasks was consistent among all the participants, so the results of the tasks cannot be regarded as being independent of each other due to learning effects. Because of the way we allocated time, only five Obsidian participants completed the Casino task, so those results should be considered exploratory.

We assessed results by manually analyzing code for correctness rather than by test cases. It is possible that using unit tests would have revealed additional bugs that we did not identify. Furthermore, disallowing execution might have biased the results to favor Obsidian, since without testing, writing correct code may be easier with the stronger type system that Obsidian provides.

Providing unit tests to participants (and allowing them to be run) might have allowed the participants to identify minor bugs that had gone undetected without incurring the time and variance cost of having participants write their own tests. This approach would also likely have increased external validity, although it adds a concern that participants might simply iterate until the tests pass rather than ensuring that they really understand how to write the code correctly. Furthermore, given our research focus on *static* correctness techniques, it is not clear which unit tests should have been included.

Our study was potentially subject to the Hawthorne effect, in which participants may change their behavior when they know they are being observed [175]. However, this effect would have been equally present in both conditions. To minimize this effect, although the experimenter remained in the room, the experimenter minimized direct observation by recording screen videos and source code rather than continuously watching participants.

11.13 Implications on PLIERS

The randomized controlled trial comparing Obsidian and Solidity served in part to evaluate Obsidian and in part to evaluate the PLIERS process (chapter 3). In this RCT, we were able to show a safety benefit of Obsidian in the Auction task, and were able to show that most of the Obsidian participants were able to use ownership successfully in the Prescription task. This shows that the tutorial method was mostly successful (though more success could likely have been obtained with more practice) and that the language design was effective overall (modulo the abuse of `disown` that we observed). Every study design involves making tradeoffs. The results here may show a tradeoff between training time and success rates; users of PLIERS will need to decide, based on their own design and research goals, how to balance the risks when designing their studies. However, the overall PLIERS design process did result in a language that had significant benefits relative to the status quo, which we were able to measure in a relatively low-cost study.

In retrospect, since only one of 20 participants completed the Casino task successfully (across both conditions), that task was too hard for the amount of time we allowed. We recommend that users of PLIERS carefully select success criteria in pilot studies in order to set appropriate task time limits and difficulties.

Designing and executing an effective RCT is extremely challenging and labor-intensive. The tutorial and tasks were initially drafted much earlier than the RCT, but even with a six-participant usability study (which had nine pilots) and four pilots for the RCT, we were still surprised at how difficult the Casino task was for some participants. In addition, executing the RCT was a lengthy effort; we estimate that recruiting participants, managing 21 participants plus four pilots, analyzing the data, etc. took about two months.

11.14 Conclusion

As programming languages are tools for empowering programmers, empirical methods offer an opportunity for designers to provide evidence of the benefits of their work. In this study, we showed that programmers who used Obsidian were able to complete more of the tasks correctly than those using Solidity (§ 11.11). We also showed that ownership alone can be used effectively with a short training period and that assets can be used to detect bugs that would otherwise likely be inserted. However, we also identified areas of risk relating to language features that weaken the checks that the compiler provides. Although few language designs have been evaluated in this way, our work shows that it is possible to empirically evaluate a novel language, to both support hypotheses of usability as well as identify areas for potential improvement. We also hope that our findings of usability for the less-common type system features we analyzed will lead to more adoption of safer, more sophisticated type systems in future languages.

	Solidity (N=6)	**Obsidian (N=8)**
How much did you like the language you used?	3.7 (0.82)	4.0 (0.53)
How well do you feel you understand the concept of ownership?	3.8 (0.98)	3.75 (0.99)
*How useful do you think ownership is?	3.0 (1.1)	4.88 (0.36)
*How well do you feel you understand the concept of states?	4.8 (0.41)	4.1 (0.64)
How useful do you think states are?	4.3 (0.81)	4.1 (0.64)
How well do you feel you understand the concept of assets?	3.2 (0.98)	3.4 (1.3)
How useful do you think assets are?	2.7 (0.52)	3 (1.2)

Table 11.9: Perceptions of ownership, states, and assets on a 1–5 scale (5 is best). Cells show average (standard deviation). * indicates that a Mann-Whitney U test shows a significant difference at $p < 0.05$.

Tasks completed correctly	**Solidity participants**	**Obsidian participants**
0	7	1
1	2	3
2	0	6
3	1	0

Table 11.10: Numbers of participants completing different numbers of tasks correctly in the two conditions.

	Solidity		**Obsidian**	
	Min	Max	Min	Max
Auction	291	355	352	395
Prescription	455	570	457	518
Casino	257	425	135	416

Table 11.11: Ranges of all (whether correct or not) solution lengths in lines of code.

Chapter 12

Study Design Challenges and Solutions[1]

In this chapter, we re-visit the PLIERS process, which was introduced in chapter 3, and consider what we learned by applying it to Glacier and Obsidian. Table 3.1 summarizes the challenges that PLIERS addresses. Our primary interest is in programmers' abilities to achieve their goals *after* they have become proficient in the programming language, not on how easy it is for novices to *learn* the language. Thus, our evaluation approach requires first teaching people a language and then observing their performance on tasks.

When we initially tried to apply HCI methods in our language design work, we were thwarted by several challenges, described in § 3.1: *training*, *recruiting*, *high prototyping cost*, and *variance*. We also encountered additional challenges, such as *interdependence of features*, *time management* in studies, *participant bias toward familiar languages*, and *unsound proposals by participants*. In this chapter, we describe techniques we used when designing the user studies described in previous chapters in order to address each challenge.

12.1 Training

Evaluating a programming language requires first *teaching* the programming language. Many universities offer term-length courses in specific programming languages or techniques; requiring this kind of time commitment would make it extremely difficult to recruit participants. Furthermore, most courses ensure a consistent experience for all students by having all students learn the material in parallel (for example, with one session per topic, where all students participate at the same time). In contrast, our design approach was iterative, consistent with design methods used in other areas of HCI [66]. We were interested in addressing a variant of our *training* challenge that asks: what would be an effective way to teach a programming language in a consistent way to many participants in sequence?

Initially, we created a textual guide to the new programming language, and asked participants to read it before doing the tasks relevant to each study. The guide was relatively short; it could be read thoroughly in under an hour. Unfortunately, this approach had very significant limitations. Although it was effective for some participants, others only skimmed the material and were then

[1] Portions of this chapter previously appeared in [45].

unable to complete the programming tasks. Because the guide was not structured as reference material and it included substantial conceptual information, skimming the guide was insufficient.

We were able to solve the problem with two adaptations: (1) break the guide into much smaller pieces; (2) ask participants to answer questions or complete small tasks to assure they had absorbed the material of each piece. For example, we broke the Obsidian tutorial into ten parts, and still the average participant completed it in under 90 minutes. We found that we were able to design tasks that checked understanding that were brief and did not require substantial experimenter intervention (helpful for ensuring consistency). We used a web survey tool (Qualtrics [158]) to guide participants through the tutorial and ask questions to check understanding. The tool also offered automatic feedback on participants' answers to multiple-choice questions. For example, Figure 12.1 shows a question about a code fragment with the correct answer selected. For another example question, see Figure 11.1. The relevant language details are explained in §6.2.

```
contract Money {
  int amount;
  transaction getAmount() returns int {
    return amount;
  }
}
contract Wallet {
  Money@Owned m;
  Wallet@Owned() {
    m = new Money();
  }

  transaction spendMoney() {
    ...
  }

  transaction receiveMoney(Money@Owned >> Unowned mon) returns Money@Owned
    Money temp = m;
    m = mon;
    return temp;
  }

  transaction checkMoney() returns Money@Owned {
    return m;
  }
}
```

Q15. Would we get a compiler error with the checkMoney function?

- ○ Yes, you cannot return a field of a contract in a transaction
- ○ No, the return type matches the type of m
- ◉ Yes, returning m makes it Unowned, which doesn't match the ownership status of m's declaration
- ○ No, m is of type Owned, which matches the ownership status of m's declaration
- ○ None of the above

Figure 12.1: One question from the Obsidian tutorial (another is shown in Figure 11.1). The question assesses whether the participant has understood that at the ends of transactions, fields must be have types that match their declarations, and that returning a variable consumes any ownership in the variable. If a participant submits an incorrect answer, the survey tool informs them of their error so they can fix their misunderstanding.

Although we originally wanted to make the tutorial stand alone so that every participant would have the same experience, we found that to be impractical; participants inevitably had questions about the materials. To see what would happen, we tried refusing to answer questions from one participant, but forcing them to continue without having their questions answered resulted in them being unable to complete the tasks. However, we found that if an experimenter was available to answer questions, most participants asked only a small number of questions, which could be addressed rapidly. This approach is arguably more similar to a real-world language learning experience than an approach in which no questions are answered; normally, learners can search the Internet for answers to their questions, ask friends for help, etc.

In summary, although our initial tutorial was not an effective way of teaching the language, and the final tutorial was not sufficient by itself, dividing the tutorial into small pieces, providing tasks to help participants check and reinforce their understanding, and having an expert who could answer questions allowed most of our participants to learn the needed material in a short period of time. Figure 11.3 shows how the tutorial was broken into 11 different sections, each of which was followed by exercises for participants to complete.

12.2 Recruiting

Evaluation requires participants who are sufficiently skilled that they can rapidly learn a new programming language and then complete tasks using the new language. This would seem to require lengthy user studies with skilled participants, who can be challenging to recruit and retain for the required period of time. *Iterative* evaluation requires a large number of participants, since participants who learned an earlier version of the language can no longer provide fresh perspectives on new ideas. Although some user interfaces for experts in other domains require recruiting members of a small population, many of those interfaces are for short-term, focused tasks rather than lengthy problem-solving tasks. Furthermore, although it is typical to conduct studies with students, this relates to our *external validity* challenge: to what extent do results from students apply to the professional software engineers that are the target of our language?

We found in our work on Glacier and Obsidian that we were able to usefully combine results from different populations. Rather than trying to exclusively obtain professional software engineers, we found that we could design studies that yield meaningful results from students; for other aspects of the research, we recruited limited numbers of professionals. For example, when we wanted to interview software engineers to find out their experiences of using immutability constructs in the Glacier work, we recruited senior-level professional software engineers. However, for the other studies, we made three observations that enabled us to do our studies with various kinds of students.

First, about 41% of professional developers have been programming professionally for less than five years [187]. Many graduate students have some professional experience. For example, students at the Professional Master's program in Computer Science & Engineering at the University of Washington were reported to have an average of five years of professional experience [143]. Similarly, the Carnegie Mellon Master of Software Engineering program requires all students to have at least two years of experience [205]. By recruiting from graduate students, we were able to attract a population that is similar to a significant fraction of professional programmers and

software engineers.

Second, in usability studies, it is typical to assume that usability problems encountered by even one user may be experienced by many others. Not every usability problem can be addressed without risking introducing new usability problems, but our experience is that many can be. For example, error messages, documentation, and keywords can be interpreted in ways that were not intended by the author; clarifying the text can prevent others from being confused in the same way. Syntax borrowed from other languages can be evocative in useful ways, but when the semantics do not match precisely, confusion can result; this can be addressed by choosing distinct syntax. On the other hand, semantic or structural changes can have consequences on users that are hard to predict, especially since one high-level change may necessitate a series of lower-level changes, which each have their own impact. For example, in Obsidian, moving transactions so that they were no longer lexically scoped in states necessitated adding special syntax for specifying the initial type of the receiver, `this`. To address this, we used a special parameter to specify the type of `this` based on consistency with Java, which already uses that design. Unfortunately, that approach was surprising to some of our participants.

By addressing problems that student participants encounter, we prevent professionals from encountering those problems as well. Of course, some of the problems may not be ones that professionals would encounter, but nonetheless, addressing them may improve learnability, making the system better overall. When changes that would improve the system for the participants might degrade performance for experienced users, then the designer can make an informed tradeoff, potentially addressing the problem in training materials rather than in a design change.

Third, for Obsidian studies, we developed a screening instrument so that we could include only participants who had appropriate programming skills. Designing a screening instrument (or deciding not to use one) depends on an assessment of what knowledge and skills are required of participants, and of how honest the prospective participants will be in their self-assessment. If prospective participants can reliably self-assess preparedness for the study, and they can be assumed to be honest, then screening may be unnecessary. On the other hand, even in this case, assessing programming knowledge and skills can be useful for understanding how these relate to task performance. In the Obsidian studies, we invited participants based on a "basic Java" portion of the screening instrument and observed that performance on one of the tasks was correlated with performance on the "advanced Java" portion of the instrument, suggesting that Java programming knowledge was a significant influence on task performance. This is somewhat surprising, since the screening test examined language-specific knowledge, but did not give any actual programming tasks. We would encourage others to consider using this kind of screening instrument, since it is low-cost (generally under 10 minutes per participant), resulted in participants who were generally capable Java programmers, and portions of it correlated with task performance.

We found that relatively small incentives were sufficient to motivate students to participate in our studies. For three-hour studies, we offered a $50 Amazon gift card; for four-hour studies, we offered a $75 Amazon gift card. For shorter studies, we paid $10/hour. We recruited professionals from among our personal networks and did not offer them a specific incentive to participate.

12.3 High prototyping cost

Programming language designers are accustomed to creating high-cost implementations, not low-cost *prototypes*, but traditional HCI methods assume that low-cost prototypes can be created. Traditional ways of evaluating programming languages typically require a compiler or interpreter as well as theoretical work to create a sound design (informally, one in which programs mean what they are supposed to mean and the safety guarantees that the type system claims to provide can actually be provided). If one insists on creating a sound, formal model of the language before evaluating it with users, iteration can require so much time that it is impractical. Furthermore, the cost is increased by the expectation of sophisticated language-dependent tooling in IDEs: syntax highlighting, autocomplete, high-quality error messages, and the like.

Instead, we do not insist on doing this work at the beginning. We outline a potentially sound underlying formalism without proving all the relevant properties. Then, we design a surface language and evaluate it with users so that we can obtain feedback early. In doing so, we accept the risk that the formal system cannot be made sound without invalidating the data we gathered from users, but in practice, we found that usually any mistakes are minor and can be corrected without having to redo the user studies.

Late in the project, we found that designing and running user studies of low-level features typically required much more time than implementing the features; for those, it make sense to implement the alternatives rather than simulating them. On other other hand, early in the project, many high-level design decisions would have required substantial design and implementation work. Among those, we carefully selected questions for which user input would be the most impactful. A key approach in minimizing cost of language changes was to re-use training materials across phases of the studies to the extent possible, allowing us to amortize the cost of their development across multiple studies. The training materials co-evolved with the implementation and represented a significant investment.

12.4 Interdependence of features

Suppose a comparison between two languages showed that one allowed participants to complete tasks faster or more successfully. If the two languages were very different from each other, it would be unclear which aspects of the new language were actually helpful. For example, a comparison between a particular functional language and a particular object-oriented language would not result in fine-grained, actionable design guidance for a new language. Furthermore, if the study was done in the context of a language that was new to participants, confusion might be due to unfamiliar aspects of the language that are unrelated to the design question of interest.

By using the *back-porting* approach described in § 3.2, we isolated particular design questions in the context of an existing language. Although this does not enable us to address very high-level design questions, such as whether the language should be object-oriented or functional, it allowed us to obtain actionable data about particular design decisions.

Theoretical refinement is another approach that helps address feature interactions, since frequently, key theoretical issues relate to interactions between language features. Likewise, case studies, natural programming, and usability studies with appropriate tasks can lead to insight

regarding cross-cutting concerns.

Of course, it is still the case that the design choices are not orthogonal. To address this, we integrate the results into a new language and conduct summative studies on the completed language as a whole as well.

12.5 High variance and external validity

The nature of programming is that there is huge variance in performance on tasks among different programmers [135]. When asking participants to complete programming tasks to help a designer iterate on a language design, participants frequently get stuck on problems that are not of interest to the designer. For example, in one Obsidian study, a participant spent significant time writing code to recurse through a data structure, even though code had been provided to do exactly that. Issues involving the details of the data structure were intended to be out of scope for the study. On the other hand, constraining tasks too much may result in artificial tasks that do not represent the complexity of real-world programming problems, which limits the external validity of the studies.

We used three techniques to address these problems. First, we combined the results of different kinds of studies (*triangulation* [172]). Qualitative studies of varied tasks with varied participants, in which timing is not an important dependent variable, can identify usability problems, and an experimenter can guide participants away from problems that are not intended to be part of the study. Quantitative studies typically involve fairly constrained tasks, but we could hope to obtain statistical significance in a comparison between two different designs. Finally, we also used case studies to address questions of *expressiveness*: elucidating what happens when the language is used to solve a larger programming problem, which cannot be completed in a single-session user study; for more information, see chapter 5 and chapter 10.

Second, particularly in RCTs (in which the experimenter cannot provide any guidance), we gave several independent tasks rather than one long task. Then, we analyzed the tasks separately, although of course the performance on the tasks is not independent because the same participant completed all of the tasks. Furthermore, dividing tasks into multiple pieces enabled separate analysis of complete vs. incomplete tasks. For example, in the Glacier studies, we gave both simple and complex immutability specification tasks rather than one combined task. In the Obsidian studies, we separated a complex task, Casino, from simpler tasks, Auction and Prescription, even though the research questions overlapped. This allowed participants to succeed in the simpler task even if the more complex task was too difficult for them. It also offered an opportunity for participants to apply knowledge gained in the simpler task when working on the more complex task.

Third, recruiting from a constrained population reduced the impact of uninteresting noise. The primary technique we used was a screening survey, which participants must complete before being selected to participate in the study. This allowed us to ensure that programmers have sufficient programming skills and knowledge. Of course, one must be careful to avoid screening out participants that may, in fact, be representative of the population to which the results should generalize.

In qualitative studies, it is sometimes unclear how many participants to recruit. Nielsen and Landauer found that the best benefit/cost ratio occurred at 3.2 participants in a set of their usability

studies, which were of a medium-large software project [137]. We found it was effective to consider the following factors in assessing when to stop recruiting more participants:

- To what extent new data (from the most recent participants) duplicates existing data
- To what extent the researcher is willing to tolerate risk of missing usability problems
- Fidelity of the current prototype (it may not be worth exhaustively testing low-fidelity prototypes)
- Specific research objectives: have the primary research questions been addressed yet?

12.6 Time management

As a practical matter, one needs to keep each participant's commitment brief in order to be able to recruit and retain enough participants and to minimize study cost. However, the experiment designer needs to allow enough time for most participants to finish the given programming tasks (at least in some of the experimental conditions). To address this problem, we conducted enough pilot studies that we could estimate the range of times that most participants would spend on each task. However, longer tasks involve more risk, as we saw with the Casino task in the Obsidian RCT. We found that we could usually allow participants enough time such that when the participant did not finish in the allotted time, the experimenter usually believed that even given substantial additional time, the participant would not have completed the task. This belief was driven by observing the difficulties that participants were facing at the end of the time window. Sometimes the problem was a design choice by the participant that made the problem much more challenging than anticipated; other times we believe it was due to lack of programming skill, since we observed some participants making basic programming errors. For example, some participants waited until writing a complete solution before compiling even once; these participants had a lot of errors, which were challenging to fix. Of course, it is difficult to generalize about participants who do not finish particular tasks when we expected them to have enough time, but we found that the above approach resulted in studies that were practical to run and which yielded useful results.

The choice of study pre-screening method introduces a tradeoff. A lax pre-screening procedure makes it easier to obtain enough participants from a population that generalizes to a broader community. A strict pre-screening procedure that admits only the most expert participants may reduce times as well as variance, but may make it difficult to recruit participants and harder to generalize the findings. In university settings, with many novices, we advise erring on the stricter end of the spectrum, since most real practitioners will be more skilled than most students.

Rather than giving fixed limits for each task in advance, we aimed to maximize effective use of participants' time. When participants had additional time remaining in their commitment (for example, in the Obsidian RCT, we told participants that the study would take four hours), we could let the participants spend longer than budgeted on the later tasks if their earlier tasks took less time than expected. Then, when reporting results, we could consider what the success rate would have been if everyone had had only the time available of the participant with the minimum time window for that task. In addition, we could report which participants succeeded given the additional time. This allowed us to make the best use of our participants' time while maintaining experimental validity.

12.7 Bias toward familiar languages

In a user study of a new programming language in which the participants are experienced programmers, one might expect that the language that performs "best" might be one with which participants are already familiar. Furthermore, when asked to join in participatory design exercises, perhaps participants might be likely to guide the design toward languages with which they are already familiar.

We used three techniques to address this problem. First, to find out what approaches might be easily learnable and would make immediate sense to participants, we adapted the *natural programming* elicitation technique [132]. In it, participants are given blank paper or a text editor and asked to write programs without being given a specific language to use. As a form of participatory design, the goal is to elicit from participants the way they would *naturally* express the ideas in question. Although traditional natural programming studies give the programmer no training at all, we took a *staged* approach. First, we asked participants to write programs on a blank screen with no training. Then, we told them information about the language design, and asked them to do additional programming tasks with the new (but still underspecified) design. For example, we gave participants a state transition diagram and asked them to write a program that expresses the state transitions. By scaffolding the participants' work in stages, we were able to answer both questions about participants' initial expectations as well as identifying what approaches might be most natural given our preliminary language design assumptions.

Second, in most of the studies, we constrained the participants' work according to our design ideas. Because the languages were designed to provide particular formal safety guarantees, we were interested in the impact of the language features related to those properties. In general, providing stronger guarantees requires that programmers enable the compiler to prove safety properties, which may require additional work from programmers. We were interested, then, in whether participants could complete tasks in the language even though they were obtaining stronger safety guarantees.

Third, we focused on observing and understanding behavior rather than preferences. By doing so, participants' prior experience was not an obstacle to overcome, but instead background we could leverage in teaching participants our language.

To encourage innovative responses (rather than ones that merely reflected prior training), we used natural programming for situations in which commonly-used languages could not directly represent the requirements we gave participants. We also used natural programming for low-level syntactic choices (e.g., keyword selection). We also instructed participants explicitly to be creative and not write in any particular existing language. Finally, we were careful to interpret the results in the context of participants' prior knowledge. For example, when participants use curly braces to denote blocks, the content of the blocks may be interesting even though the choice of curly braces is not.

12.8 Unsound proposals by participants

Another common limitation of natural programming is that participants lack expertise in language design, resulting in unsound proposals. This problem occurs with participatory design in other

domains as well, and the usual solution is to use participant ideas as input to an expert-led design process [150], which applies here as well.

Our language design process typically involves writing multiple example programs, each of which assumes a particular language design and explores a particular kind of programming problem. The examples typically expose tradeoffs in language design; choosing which tradeoff to make can be informed with user input. We were able to use some of these prototypes to develop user studies, in which we presented participants with several options rather than expecting them to compose designs from scratch. In some cases, we tried to generate all feasible options in a particular design space (due to various technical constraints, this might result in three or four options), and then narrowed this down to the most promising approaches based on the tradeoffs that were apparent. We asked participants to *complete tasks* using the best candidates so that we could come to an informed conclusion about which of the options were best, rather than merely asking participants for their opinions. This allowed us to focus the process on designs that would fulfill the technical requirements while still obtaining relevant design insights.

Chapter 13

Future Work and Conclusion

13.1 Future Work

There are three main directions in which I would like to proceed with this research. First, *methods* (§ 13.1.1): how can we further develop user-centered language design techniques so that they are more easily applied, give more externally-valid results, apply to more kinds of situations, and can be applied by more kinds of language designers? In the course of doing this work, I developed both language design and usability expertise myself, and was therefore able to do both parts of the process. However, the methodology would be more useful if it could be applied with only one of those two kinds of expertise (and more useful still if domain-specific languages could be developed with only domain expertise). Second, *applications* (§ 13.1.2): the language design methodology should be broadly applicable in domain-specific language and general-purpose languages; in languages for novices and in languages for experts. Finally, *technology* (§§ 13.1.3 and 13.1.4): develop type and verification systems that provide strong safety guarantees, guided by insights from user research. For example, how can we make domain-specific specifications more easily verifiable?

Novel type system designs have, in the past, taken many years to achieve adoption. I hope that with PLIERS, safer languages can be designed in a way that helps people be as effective as possible at achieving their goals. These successes will hopefully motivate faster adoption of safer programming techniques, improving the quality of software across all contexts in which people create software.

Although this dissertation shows particular language designs that are helpful for users, much work remains to develop techniques that help show the impact of language designs in the real world. For example, although specifying which classes are immutable avoids bugs in which programmers accidentally mutate immutable structures, are there other costs of the additional language complexity? Perhaps more importantly, given that immutability may have other costs (e.g., performance, API complexity), should developers use immutability in the first place?

It is not practical for researchers to randomly assign real engineering teams to use a design or not (immutable data structures or interfaces, for example). Instead, one could try to find *natural experiments* in which external events drove adoption of particular designs. Natural experiments [52] have been used in other fields, such as economics, in which researchers leveraged changes

that the researchers did not control to make causative inferences regarding effects. Researchers in HCI have used these techniques on programming languages for novices [57]; I would like to try to find natural experiments that inform professional software engineering practice as well.

13.1.1 Methods

I evaluated PLIERS by applying it to two different language designs. This approach greatly facilitated developing and iterating on the PLIERS process. However, in the future, I would like to show that these methods can be used by language designers with a variety of backgrounds and goals. As a practical matter, recruiting language designers to participate in a language study is a challenging and heavyweight endeavor, but future work may identify promising contexts in which to evaluate PLIERS more broadly. I have begun by teaching PLIERS to students in an undergraduate programming language design class, and found that the students were able to use some of the methods to help them iterate on their language designs.

Currently, there is a gulf separating programming language designers from usability experts. Typically, language designers and usability experts have different skills and interests. Although I have learned and applied both kinds of expertise in my work, many people prefer to focus on either the theory of programming languages or the methods of software engineering or HCI research. What I have found in my work is that programming language design benefits from *all* of these perspectives. I encourage the community to consider two models of addressing this problem. First, it could become standard that all new languages for people are the results of collaboration between usability experts and programming language experts. Or, we could enable those who have expertise in one area to use tools that aid them in the other area:

- The theoretical aspects of the design work require substantial background. Perhaps in the future, mechanized tools could help those who are not programming language experts design safe languages in their own application domains. By using program synthesis techniques, a language synthesis tool might be able to search the space of languages that have particular formal properties to help users identify safe design candidates. Creating Obsidian required spending months developing the underlying core calculus and proving it sound; if some of this effort could be mechanized, language iteration would be much faster, and could be feasible for those without formal programming language training. Although I spent several months working on mechanizing a soundness proof of Silica in Agda [26], the process was extremely slow, and I only finished a proof of soundness for a tiny fragment of the language. Part of this time was spent learning Agda, but nonetheless, this is suggestive of a need for better tool support.
- Running experiments is challenging and high-cost. What if some of the usability methods needed for developing languages could be packaged in a re-usable way for convenient application? A mechanism for running experiments with appropriate participants at appropriate scale might make it feasible for designers to study the usability implications of different designs without having to become expert in experimental design methodology.

The Wizard of Oz approach that I proposed relies on an experimenter who can accurately simulate the kinds of error messages that a compiler might generate. I envision a utility that would help promote consistency and improve reliability. Such a tool would accept error messages

entered by experimenters in real time during experiments and deliver them to participants in a realistic way. By recording the errors that were delivered, the experimenter could re-use existing error messages as well as record participants' reactions. Although this would not obviate the need for a well-informed experimenter, the approach might lead toward refined error messages that are clearer for users to understand.

The tradeoff between internal and external validity seems fundamental to programming studies. Above, I suggested increasing external validity, but doing so might compromise internal validity further, since the tasks would include more kinds of work. Another approach to study design might trade external validity for additional internal validity. Asking participants to write nontrivial programs, such as in the Casino task, may reflect real-world programming tasks, but the task complexity results in high variance and significant amounts of time spent on programming issues that are not necessarily directly related to the research questions. In future studies, it might be worthwhile to consider more restricted tasks that only investigate a small portion of the development workflow. For example, if the research question pertains to creation of interfaces or architectures, then the task might isolate that part of the programming process rather than integrating it into a complete programming problem. Another approach to increasing external validity would be to investigate whether the results generalize to other languages or contexts. For example, do Java programmers who write auction programs also tend to accidentally lose assets? How does task performance compare between Obsidian and other linearly-typed languages, such as Nomos [56]?

Another limitation of the methods of PLIERS is that although one can do studies that assess the usability of particular language design choices, in some cases design choices interact with each other. As a result, it is not clear that designers can combine the results of different studies and expect that the resulting language will be usable. In this work, I mitigate this threat in two ways. Summative studies address the integrated language and can reveal problems that arise from combinations of design choices. Likewise, by triangulating design through multiple kinds of studies, some of which crosscut multiple language design choices, I obtain different perspectives on various combinations of features. However, future method development work may be able to address this problem more directly.

13.1.2 Applications

In the future, I hope to explore how PLIERS could be used to develop tools for other problem-solving contexts beyond programming, and for other kinds of programming languages. For example, robotics engineers frequently develop software that implements state machines, but they do so in general-purpose languages. Languages for teaching programming offer a particularly convenient opportunity, since programming courses may offer large groups of potential participants in studies, and learnability is particularly easy to measure. In contrast, when the end goal is to improve long-term performance of professionals, one must either conduct longitudinal studies or extrapolate from short-term studies.

Attributes of programming that are shared with other kinds of problem-solving activities include:

High variance: Problem-solving can be unpredictable [119]; in user studies, some participants

typically complete tasks almost instantly whereas others can spend hours working and still not finish. This large variance makes running quantitative user studies very challenging.

Range of working styles: Bergström and Blackwell described a diverse collection of different approaches to programming problems [14], such as *bricolage/tinkering* and *engineering*. These different styles may be used even by different people using the *same language*, impeding a designer's attempts to anticipate a user's strategy or behavior.

High stakes: Errors when programming can contribute to serious real-world safety problems, e.g., in avionics or health care systems.

For example, CAD tools affect their users' creative processes [171]; likewise with process engineering tools [30] and even drug design tools [193]. All of these domains involve expert problem-solving by a variety of different people with high costs of failure. As such, they might be amenable to use the PLIERS process to help designers in those domains create tools that are effective for their target users.

13.1.3 Glacier

Future work on Glacier should expand the range of situations to which Glacier applies by adding support for delayed initialization of fields (for example, caches) in immutable objects and support for initialization of circular data structures. In addition, Glacier does not consider external sources of mutability, such as the filesystem or network; future work should analyze to what extent these kinds of hidden mutability compromise the guarantees that Glacier provides. A future corpus study could analyze to what extent the system applies to existing code. A refactoring tool could help software engineers adopt Glacier more easily and also be used in a corpus study of applicability.

I have not found data regarding to what extent (and in what situations) designing components to be immutable is beneficial. Understanding when to make components immutable is a critical step in using immutability systems effectively.

13.1.4 Obsidian

The Casino task of the RCT (chapter 11) , which included significant use of nominal states (i.e., statically-defined states with their own names), resulted in Obsidian participants writing dynamic checks. Future work should investigate the extent to which typestate, as provided in Obsidian and other typestate-oriented languages, can be made more compelling for programmers. For example, when pairs of objects have states that are coupled, the language could provide features to make representing and using this relationship convenient. Likewise, the existence of risk compensation among programmers should be investigated in future studies.

Although the study showed that Solidity programmers insert bugs that Obsidian detects, a corpus study could show how prevalent these kinds of bugs are in real-world Solidity code. Bugs involving asset loss can occur in any language in which programs may manipulate assets; a corpus study involving a larger corpus of programs might be fruitful and still generalize to smart contract development. Indeed, it may be that the kinds of safety properties needed in smart contract development are generally applicable to many kinds of programs, regardless of whether they are hosted on blockchains.

A study that included testing, debugging, and code review would be more representative of real-world use. Also, the participants were new to the language they used in the study, and the Casino task may have been too difficult for the amount of time I allocated. A longer-duration study would likely have increased validity — both by allowing enough time for difficult tasks and by allowing participants to become more comfortable with the language they were using. In a study of experienced Obsidian and Solidity programmers, I hypothesize that the task completion time difference would diminish significantly but Obsidian users would continue to take longer on complex development projects. On the other hand, there would likely be fewer serious bugs in the completed Obsidian projects, since Obsidian rules out classes of important bugs.

Some of the usability results for Obsidian may generalize to other languages that use ownership, such as Rust. Future work should investigate whether the usability tradeoffs I observed here occur in other languages.

I observed many Obsidian programmers making unsafe use of `disown`. Future research should investigate how to more safely provide features that are safe in unusual situations but *unsafe* in situations that arise commonly. Some languages use naming conventions, such as "unsafe," to denote features that defeat the language's type system (e.g., Haskell's `System.IO.Unsafe.unsafePerformIO`), but here, the feature is a required part of the type system rather than a way of escaping from it, so giving it a dangerous-sounding name may not be a good solution.

13.2 Conclusion

In this dissertation, I have shown through the PLIERS process and its application to two different language designs how user-centered design methods are useful in the design of programming languages. They can be used to address a host of design and research questions, such as:

- What challenges do users of existing languages face, and how common are they?
- How do users naturally express solutions to particular problems, given particular constraints on the solution space?
- What names do users give to particular concepts?
- What barriers to success face particular kinds of users when they try to do specific tasks?
- Which aspects of a language design do users find confusing?
- Which aspects of a language design are error-prone?
- Can particular kinds of users complete specific tasks without getting stuck?
- How long does it take for users to do specific tasks?
- In which language (of several concrete options) can users who complete tasks typically complete tasks faster?
- How likely are users (in a lab study) to insert particular bugs when using tools that do not guard against them?
- How long does it take for users to learn to use particular new language features?

These methods can be used at practical cost (even in a university environment, without ready access to large numbers of professional software engineers) to obtain language design insights

and evidence of language design success that are relevant to at least some professional software engineers. PLIERS couples use of the above methods with a theoretical perspective, which enables development of languages that have well-understood semantics and safety properties. It also enables drawing inspiration for language design from the theory of programming languages, which has long been a source for interesting language design ideas. At the same time, the creativity of the language designer is preserved; input from users serves to guide language refinements and identify challenges, but not to actually determine the final design.

A key form of leverage that user-centered methods provide is the ability to help the designer focus on design choices that result in a *simple* language that is as effective as possible for users, rather than the most *expressive* language possible. Although it may be tempting for a designer to develop a very expressive language, feedback from users can help identify usability challenges in parts of the language that may not be necessary for the most important use cases for the language. Then, the designer can apply their own insight and expertise to modify the language to address those challenges.

In Glacier, I showed how interview studies can be used to narrow the design space to a small subspace that shows promise. Then, I showed how qualitative user studies can be used to identify usability problems in a preliminary design. Finally, I showed how quantitative studies can be used to show in a lab setting that (a) a novel language feature is easier to use effectively than the existing language features; (b) for a particular kind of task, users of the existing language tend to frequently insert bugs that the type system I designed rules out. It is important to show both of these facts together; showing only usability of a feature intended to improve safety might not lead to real benefits, since users might not actually write code that violates the safety constraints. Likewise, showing only that people tend to insert bugs that the language rules out does not imply that people can use the language feature successfully.

Glacier shows that choosing a *simple* design can in many cases be much more beneficial for users than choosing a *very expressive* design. Expressivity typically has a complexity cost, and the increased complexity can lead to increased learning costs. Although the studies I did did not try to compare error rates between Glacier and IGJ-T, I expect that more complex designs would lead to higher error rates, as the additional complexity leads to higher cognitive load. This kind of comparison would be valuable future work, as it might lead toward a theoretical framework for analyzing usability implications for design decisions.

In Obsidian, I showed how a wider variety of formative methods can be adapted for effective use in the design of programming languages, and how the process of applying formative methods to programming language design can lead to insights regarding the methods. Then, I showed how summative studies can show situations in which the new language can be used effectively as well as potential challenges for effective use. For example, back-porting proposed designs to existing languages makes recruitment easier and removes confounding variables, allowing researchers to focus the study on the particular language construct in question. Likewise, Wizard of Oz studies can be applied to programming languages that do not have tools built yet so that researchers can obtain early feedback. Finally, a quantitative lab study showed that Obsidian addresses bugs that can occur frequently in typical development tasks. In the lab study, most participants were able to use Obsidian's ownership system to statically rule out classes of bugs in their tasks.

At the same time, I showed how to combine typestate with linear assets in a language. I proved a type soundness theorem for Silica, which forms a foundation for Obsidian. As a corollary,

among all references to a given object that exist at once, at most one is an owning reference. Likewise, I proved an asset retention theorem, showing that if evaluation takes a step and a previously-owned object no longer has an owner, the only possible statement that was evaluated was `disown`.

It is important to consider the cost/benefit tradeoff in using user-centered methods to design programming languages. Although I wrote about the benefits of user-centered methods in the design of Obsidian and Glacier, these benefits came at a significant cost in time and participant-recruiting expenses. Because my research question was about how to use user-centered design methods to design programming languages, I applied the methods to address a wide variety of questions. For some of those questions, in retrospect, it might have been more expedient to either obtain the insight by personal experimentation or to make an arbitrary choice, since the impact of the user study was not very large. For example, I found that the natural programming approach did not typically yield good keyword suggestions; many suggestions already had other meanings that did not match the needed one, and in many cases, all of the participants had different suggestions. Likewise, although conducting a natural programming study regarding use of state machines showed that some people did use them, it is not clear that *naturalness* implies *usability* or that lack of naturalness implies lack of usability. This relationship is worthy of further investigation, but as a practical matter, it may be more worthwhile to focus more on usability and less on naturalness.

Designing and running user studies is very time-intensive; it can take a month or more to design even a short user study to answer a narrow question. If the same question can be answered by the designer by doing prototyping or case study work, that is likely more cost-effective. Of course, if a study is to be done, it is best done *early*, while significant design changes are still possible. This consideration motivated much of my work on user studies with incomplete prototypes.

On the other hand, user studies can reveal serious usability problems that need to be addressed. The permissions user study (§ 8.2.3) identified several serious problems, such as with ownership transfer, which I was able to address with both changes to the language and changes to the learning materials. It can be hard to distinguish between the impact due to materials changes and the impact due to language design changes. However, a language design change can enable a clearer or simpler explanation, so the two are intimately intertwined. This limitation may be fundamental to the method; it is not clear how one would design a study whose results do not depend on the training materials.

My advice for those looking to use user-centered design methods is to be judicious. User studies can be helpful when:

- The design involves new concepts and there is a risk of users not being able to use or understand them effectively
- Several design alternatives seem equally attractive, and the decision is a significant one in the overall design
- Only limited data exist regarding the usability implications of key design choices

To what extent do user-centered methods result in *high-level* design changes to the language, as opposed to low level changes such as keyword choice? The answer depends on where the user-centered methods are deployed. If one does a usability study of low-level questions, then one can expect low-level answers. But a user study of high-level design choices can reveal major problems, which can prompt re-visiting high-level design choices. Usability studies generally do

not themselves propose solutions; they identify problems. From there, it is up to the designer to decide how to address the problems. Some problems can be addressed with low-level changes, in which case those are likely the best solution. But if the problems are more fundamental, and the designer is not able to address them via low-level changes, then higher-level changes may be the best solution.

It is traditional for researchers to explore how to create more powerful, expressive designs. This can certainly be useful for expanding the boundaries of what languages can be created. However, in this research, I have shown how starting from a *user* perspective rather than a technical perspective can also result in technical innovation. In many cases, designing a system that is simple from a user's perspective can present challenging technical problems. However, when these are solved, the result can be very effective for users. I hope that other researchers will take this perspective, since it facilitates development of tools that are better-matched with the needs of users.

Appendix A

Auxiliary Judgements for Silica

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:

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

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.

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

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.

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

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.

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

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

$$\begin{array}{rl} transactionName\ (Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST} x})T_{ST} \gg T_{ST}) & \triangleq m \\ transactionName\ (Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST} x})T_{ST} \gg T_{ST}\ e) & \triangleq m \\ transactionName\ (\overline{T_{ST} >> T_{ST} f} Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST} x})T_{ST} \gg T_{ST}) & \triangleq m \\ transactionName\ (\overline{T_{ST} >> T_{ST} f} Tm\langle\overline{T_M}\rangle(\overline{T \gg T_{ST} x})T_{ST} \gg T_{ST}\ e) & \triangleq m \end{array}$$

$$transactionNames\ (\overline{M}) \triangleq \overline{transactionName\ (M)}$$

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

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

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

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

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

$$\frac{}{stateName\,\left([\textsf{asset}]\ S\ \overline{F}\right) = S} \qquad \frac{states\,(D) = \overline{S}}{stateNames\,(D) = \overline{stateName\,(S)}}$$

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

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

$$\frac{def\,(C) = \textsf{contract}\ C\langle\overline{T_G}\rangle\ \textsf{implements}\ I\langle\overline{T}\rangle\{\overline{ST}\ \ \overline{M}\}}{params\,(C) = \overline{T_G}}$$

$$\frac{def\,(I) = \textsf{interface}\ I\langle\overline{T_G}\rangle\{\overline{ST}\ \ \overline{M_{SIG}}\}}{params\,(I) = \overline{T_G}}$$

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

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

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

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

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

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

Auxiliary Judgment 12: $\overline{\textsf{S}}$ **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: $\overline{\mathrm{S}}$ **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: $possibleStates_\Gamma\,(T_C@T_{ST}) = T_{ST}$

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash nonAssetState\ \left(possibleStates_{\Gamma}\ \left(D\langle\overline{T}\rangle@T_{ST}\right)\right)}{\Gamma \vdash nonAsset\ \left(D\langle\overline{T}\rangle@T_{ST}\right)} \qquad \frac{X@p\ \mathsf{implements}\ I\langle\overline{T}\rangle@T_{ST_i} \in \Gamma}{\Gamma \vdash nonAsset\ (X@T_{ST})}$$

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

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.

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

Auxiliary Judgment 21: $\mathit{notOwned}\ (\mathbf{T})$

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

Auxiliary Judgment 22: $\mathit{maybeOwned}\ (\mathbf{T})$

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

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 $I\langle\overline{T}\rangle.\mathsf{Owned}$. However, a concrete type such as $C\langle\overline{T}\rangle@S$ is already as specific as possible—therefore, its bound is itself.

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

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

Auxiliary Judgment 24: $\Gamma \vdash \mathit{bound}_*\,(T_{ST})$

$$\frac{P \in \{\mathsf{Owned}, \mathsf{Shared}, \mathsf{Unowned}\}}{\Gamma \vdash \mathit{bound}_*\,(P) = P} \qquad \frac{}{\Gamma \vdash \mathit{bound}_*\,(\overline{S}) = \overline{S}}$$

$$\frac{[\mathsf{asset}]\ X@p\ \mathsf{implements}\ I\langle\overline{T}\rangle@T_{ST} \in \Gamma}{\Gamma \vdash \mathit{bound}_*\,(p) = T_{ST}}$$

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

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

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

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: $Var\,(T_G)$, $PermVar(T_G)$, $Perm(T)$

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

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

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

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

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

To check $implementOk_\Gamma\,(I\langle\overline{T}\rangle, S)$, 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: $\mathit{subsOk}_\Gamma\left(T, T_G\right)$

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

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

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 $X@\mathsf{Owned}$, 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: $\mathit{genericsOk}_\Gamma\left(T_G\right)$

$\mathit{genericsOk}_\Gamma\left(T_G\right)$ expresses whether a use of a type parameter is suitable when the parameter must implement a particular interface.

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

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

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

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

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

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

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

$\mathit{specializeTrans}_\Gamma\left(m\langle\overline{T_M}\rangle, D\langle\overline{T}\rangle\right)$ 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 $\overline{T_G}$.

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

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 $d ::= x \mid x.f$.

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

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

$T \oplus T \triangleq T$

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

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

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

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

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

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.

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

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

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

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.

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

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

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

A.3 Soundness Theorems

Theorem A.3.1 (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****,* $\Sigma, e \to \Sigma', e'$ *for some environment* Σ'*.*
3. *e is stuck at a bad state transition — that is,* $e = \mathbb{E}[l \to_{Shared} S(\overline{s})]$ *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. 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 $b \to \Sigma_\rho(l)$ by rule E-lookup.

Subcase: $b = o$. Then b is a value.

Case: T-let. Because e is closed, $e =$ let $x : T = e_1$ 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, Σ, let $x : T = v$ in $e \to [\rho[l \mapsto v]/\rho]\,\Sigma, [l/x]e$.

Case: $\Sigma, e_1 \to \Sigma', e_1'$**.** Then E-letCongr applies, and $\Sigma, e \to \Sigma'$, let $x : T = e_1'$ in e_2.

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

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

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

Case: T-new. Because e is closed, $e = \mathsf{new}\ C\langle\overline{T}\rangle@S(\overline{l})$ (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 (9.4.2), $\overline{l} \in dom(\rho)$, and $\overline{\rho(l)}$ is well-defined. By E-new:

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

.

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 $1 \le i \le |\overline{s'}|$ by well-typedness of $s.f_i$ and global consistency. By rule E-field, $\Sigma, s.f_i \to \Sigma, s'_i$.

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, $\Sigma, l.f_i := l' \to [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(l''_1, l''_2, \ldots, l''_{i-1}, l', l''_{i+1}, \ldots, l''_{|l''|})]/\mu]\,\Sigma, ()$.

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, $\Sigma, e \to \Sigma'', \boxed{\overline{[l'_1/x]}[l'_2/\mathsf{this}]e'}^{\rho(l_1)}$

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-$\to_p$. Because e is closed, $e = l \to_p S(\overline{l'})$. 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-$\to_{owned}$, $\Sigma, l \to_{owned} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(\overline{l'})]/\mu]\,\Sigma, ()$

Subcase: $T_{ST} = Shared$**.**

Case: $\rho(l) \notin \phi$**.** Then by E-$\to_{shared}$,

$\Sigma, l \to_{shared} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T'}\rangle@S(\overline{l'})]/\mu]\,\Sigma, ()$.

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
$e = \mathbb{E}[l \rightarrow_{\mathsf{Shared}} S(\overline{l'})]$ where $\mu(\rho(l)) = C^*\langle\overline{T^*}\rangle@S'(\ldots)$, $S \neq S'$, $\rho(l) \in \phi$, and $\Gamma; \Delta \vdash_s l : C\langle\overline{T'}\rangle.\mathsf{Shared} \dashv \Delta'$. $C = C^*$ due to memory consistency.

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

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

Case: T-assertPermission. Because e is closed, $e = [\,l@T_{ST}\,]$. By rule E-assert, Σ, assert l 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, $e =$ if l in$_{owned}$ S then e_1 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, $e =$ if l in$_{shared}$ S then e_1 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 $l : C\langle\overline{T'}\rangle.\mathsf{Shared}$.

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, $e =$ if l in$_P$ p then e_1 else e_2. By assumption Γ, Σ, Δ **ok**, so $\xi(p) = T_{ST}$ for some T_{ST}. Then $\Sigma,$ if l in$_P$ p then e_1 else $e_2 \rightarrow \Sigma,$ if l in$_P$ T_{ST} then e_1 else e_2.

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

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

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

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

Case: T-pack. By *pack*, $\Sigma,$ 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, $\Sigma, \boxed{\mathrm{v}}_o \rightarrow [(\phi \setminus o)/\phi]\,\Sigma, v$. 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, $\Sigma, \boxed{e'}_o \to \Sigma', \boxed{e''}_o$. 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, $\Sigma, \boxed{\text{v}}^o \to [(\psi \setminus o)/\psi]\,\Sigma, v$. Otherwise, by the induction hypothesis, either $\Sigma, e' \to \Sigma', e''$, or e' is stuck with an appropriate evaluation context. In the former case, by E-box-ψ-congr, $\Sigma, \boxed{\text{e'}}^o \to \Sigma', \boxed{\text{e''}}^o$. In the latter case, e is stuck with an appropriate evaluation context.

□

Theorem A.3.2 (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'*,* $\Gamma'; \Delta' \vdash_s e' : T' \dashv \Delta'''$*,* $\Gamma', \Sigma', \Delta'$ ***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: $T = \mathsf{unit}$

By assumption, Γ, Σ, Δ **ok**, and $\Gamma; \Delta \vdash_s l : T \dashv \Delta''$. By assumption and E-Lookup, $\Sigma, l \to \Sigma, \Sigma_\rho(l)$. The fact that $\Sigma_\rho(l) = ()$ follows directly from global consistency. Then by T-(), $\Gamma; \Delta \vdash_s () : \mathsf{unit} \dashv \Delta$. 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 (A.3.12). 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. $\Sigma, l := l' \to [\rho[l \mapsto o]/\rho]\,\Sigma, ()$
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. $\Gamma \vdash \textit{disposable}\ (T_l)$

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

By assumption, $\Gamma; \Delta \vdash_s \textsf{new}\ C\langle\overline{T'}\rangle@S(\overline{l}) : C\langle\overline{T'}\rangle@S \dashv \Delta''$; also, $e' = o$, and $o \notin dom(\mu)$. Let $\Delta' = \Delta'', o : C\langle\overline{T'}\rangle@S$. By *T-lookup*, $\Gamma; \Delta'', o : C\langle\overline{T'}\rangle@S \vdash_s o : C\langle\overline{T'}\rangle@S \dashv \Delta'', o : C\langle\overline{T'}\rangle@Unowned$. 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 $\Gamma \vdash T <: stateFields(C\langle\overline{T'}\rangle, S)$, 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 = let $x : T_1 = v$ in e_2 By assumption:

1. Σ, let $x : T_1 = v$ in $e_2 \to [\rho[l \mapsto v]/\rho]\,\Sigma, [l/x]e$
2. Γ, Σ, Δ **ok**
3. $\Gamma; \Delta \vdash_s$ let $x : T_1 = v$ 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) $\Gamma \vdash$ *disposable* (T_1')
   (d) $l \notin dom(\rho)$
2. Let $\Delta' = \Delta^*, l : T_1$. By the substitution lemma (A.3.13) 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 $refTypes(\Sigma, \Delta, o)$.
   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 (A.3.12), 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. $\Gamma; \Delta \vdash_s () : \textsf{unit} \dashv \Delta$
2. $\Gamma; \Delta, x : \textsf{unit} \vdash_s e_2 : T \dashv \Delta^*, x : T_1'$
3. $\Gamma \vdash$ *disposable* $(\textsf{unit})$
4. $l \notin dom(\rho)$

Let $\Delta' = \Delta^*, l : \textsf{unit}$. By the substitution lemma (A.3.13) $\Gamma; \Delta^*, l : \textsf{unit} \vdash_s [l/x]e_2 : T \dashv \Delta^{**}, l : T_1'$. 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 = let $x : T_1 = e_1$ in e_2.

1. By assumption:
   (a) Γ, Σ, Δ **ok**
   (b) $\Gamma; \Delta \vdash_s$ let $x : T_1 = e_1$ 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) $\Gamma \vdash \textit{disposable}\,(T_1')$
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 A.3.3.2 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, $\Gamma{*}{*}; \Delta' \vdash_s$ let $x : T_1 = e_1'$ in $e_2 : T \dashv \Delta^{***}$, where $\Delta^{***} <^l_{\Sigma'} \Delta''$.
6. By A.3.15, $\Gamma^{**}, \Sigma', \Delta'$ **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) $\Sigma, l_1.m\langle\overline{M}\rangle(\overline{l_2}) \to [\psi, \rho(l_1)/\psi]\,\Sigma'', \boxed{\overline{[l_2'/x]}[l_1'/\mathsf{this}]e}^{\rho(l_1)}$
   (b) Γ, Σ, Δ **ok**
   (c) $\Gamma; \Delta_0, l_1 : C\langle\overline{T}\rangle@T_{STl1}, \overline{l_2 : T_{l2}} \vdash_s l_1.m\langle\overline{M}\rangle(\overline{l_2}) : T \dashv \Delta_0, l_1 : T_{l1}', \overline{l_2 : T_{l2}'}$
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) $T_{l1}' = funcArg\left(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T_{this}'\right)$
   (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 (A.3.9), 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 $\overline{l_2 : C'\langle\overline{T'}\rangle@T_{ST}'}$, by global consistency.

4. Let $\Gamma' = \Gamma, \overline{T_D}, \overline{T_M}$. By the substitution lemma (A.3.13) on 3a, we have:
$\Gamma'; l_1' : C\langle\overline{T}\rangle@T_{this},\ \overline{l_2' : C'\langle\overline{T'}\rangle@T_x} \vdash_{s_1} [l_2'/x][l_1'/\mathsf{this}]e : T \dashv l_1' : C\langle\overline{T}\rangle@T'_{this}, \overline{l_2' : C'\langle\overline{T'}\rangle@T_{xST}}$
5. *funcArgResidual* is defined in AJ:35. Let:

$$\begin{aligned} T_{l1R} &= \mathit{funcArgResidual}\left(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this}\right) \\ \overline{T_{l2R}} &= \overline{\mathit{funcArgResidual}\left(T_{l2}, T_x, C_x@T_{xST}\right)} \\ \Delta' &= \Delta, l_1 : T_{l1R},\ \overline{l_2 : T_{l2R}}, l_1' : C\langle\overline{T}\rangle@T_{this}, \overline{l_2' : C'\langle\overline{T'}\rangle@T'_{l2}} \end{aligned}$$

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

$$\begin{aligned} T_{l1R} &= \mathit{funcArgResidual}\left(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this}\right) \\ T'_{l1} &= \mathit{funcArg}\left(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this}\right) \end{aligned}$$

If $T'_{l1} \neq C\langle\overline{T}\rangle@T'_{this}$, there are two possibilities, both with $C\langle\overline{T}\rangle@T_{this} = \mathsf{Unowned}$. If $C\langle\overline{T}\rangle@T_{STl1} = T_C@\mathsf{Shared}$, then $T_{l1R} = T_C@\mathsf{Shared}$; 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 $p \in \mathit{PermVar}\,(\Gamma)$. Note that ξ' additionally contains mappings for each $\overline{T_G}$ and $\overline{T_D}$, so $\mathit{PermVar}\,(\Gamma') \subset \{p \mid \xi(p) = T_{ST}\}$, 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 $T_{l1R} = \mathit{funcArgResidual}\left(C\langle\overline{T}\rangle@T_{STl1}, C\langle\overline{T}\rangle@T_{this}, C\langle\overline{T}\rangle@T'_{this}\right)$. 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 $T_{this} = \mathsf{Unowned}$, which is compatible with all other references.
   - **Case: FuncArg-shared-unowned.** Previously, $l_1 : C\langle\overline{T}\rangle@\mathit{Shared}$ was in Δ, and Γ', Σ, Δ **ok**. Now, Δ' includes both $C\langle\overline{T}\rangle@T_{this}$ and $C\langle\overline{T}\rangle.\mathsf{Shared}$. But $T_{this} = \mathsf{Unowned}$, 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 $C\langle\overline{T}\rangle.\mathsf{Unowned}$. But $\mathsf{Unowned}$ 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. $e = \mathsf{if}\ x\ \mathsf{in}_{owned}\ T_{ST}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ because e is closed.

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_s \text{if } l \text{ is in}_{\text{owned}}\ S \text{ then } e_1 \text{ else } e_2 : T_1 \dashv \Delta''$
   (c) $\Sigma, \text{if } l \text{ is in}_{\text{owned}}\ S \text{ then } e_1 \text{ else } e_2 \to \Sigma, e_1$
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 A.3.19, 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. $e = \text{if } l \text{ is in}_{\text{shared}}\ \overline{S} \text{ then } e_1 \text{ else } e_2$

1. By assumption, and because e is closed:
   (a) $\Gamma, \Sigma, \Delta_0, l : C\langle\overline{T}\rangle@Shared$ **ok**
   (b) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Shared \vdash_s \text{if } l \text{ is in}_{\text{shared}}\ S \text{ then } e_1 \text{ else } e_2 : T_1 \dashv \Delta''$
   (c) $\Sigma, \text{if } l \text{ is in}_{\text{shared}}\ S \text{ then } e_1 \text{ else } e_2 \to [\phi, \rho(l)/\phi]\,\Sigma, \boxed{e_1}_{\rho(l)}$
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) $\Gamma \vdash bound\,(T_{ST}) \neq \mathsf{Unowned}$
   (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 A.3.19, 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. $e =$ if l is in$_p$ $\overline{S}$ then e_1 else e_2

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) Σ, if l is in$_p$ S then e_1 else $e_2 \to \Sigma, e_2$
   (c) $\Gamma; \Delta \vdash_s$ if l is in$_p$ S then e_1 else $e_2 : T_1 \dashv \Delta''$
2. By inversion:
   (a) $\mu(\rho(l)) = C\langle\overline{T}\rangle@S'(\ldots)$
   (b) $S' \notin \overline{S}$
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), 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 A.3.19, 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 A.3.19, 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) $\Gamma; \Delta \vdash_s$ if l is in$_{\text{Perm}}$ p then e_1 else $e_2 : T_1 \dashv \Delta''$
   (c) Σ, if l is in$_{\text{Perm}}$ p then e_1 else $e_2 \to \Sigma$, if l is in$_{\text{Perm}}$ T_{ST} then e_1 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) Perm $= toPermission\,(T'_{ST})$
3. In order to perform substitution for type parameters, we must have proved $\mathit{subsOk}_\Gamma\,(T, T_G)$, so we must have $\Gamma \vdash T_{ST} <:_* p$. Then by 2b and the permission variable substitution lemma A.3.10, 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 $P = \mathsf{Unowned}$, then $T'_{ST} = \mathsf{Unowned}$, and by 2c we can apply T-IsIn-Unowned to show $\Gamma; \Delta, l : T_C@\mathsf{Unowned} \vdash_s$ if l is $\text{in}_{\text{Unowned}}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.

If $P = \mathsf{Shared}$, then $T'_{ST} = \mathsf{Shared}$, and by 2c we can apply T-IsIn-Dynamic to show $\Gamma; \Delta, l : T_C@Shared \vdash_s$ if l is $\text{in}_{\text{shared}}\ \overline{S}$ then e_1 else $e_2 : T_1 \dashv \Delta'_f$.

If $P = \mathsf{Owned}$, then $\Gamma \vdash T'_{ST} <: \mathsf{Owned}$, so *maybeOwned* $(T_C@T'_{ST})$, and $\Gamma \vdash \overline{S_x} <:_* T'_{ST}$, where $\overline{S_x} = possibleStates_\Gamma\,(T_C@T_{ST})$. Then by the subtype substitution lemma lemma A.3.3.1 and by 2c we have $\Gamma; \Delta, l : T_C@\left(\overline{S_x} \setminus \overline{S}\right) \vdash_s e_2 : T_1 \dashv \Delta''$. Now we can apply T-IsIn-StaticOwnership to get $\Gamma; \Delta, l : T_C@T'_{ST} \vdash_s$ if l is $\text{in}_{\text{owned}}\ \overline{S}$ 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. Σ, if l is in_{P} Perm then e_1 else $e_2 \to \Sigma, e_1$

By inversion:

1. Perm $\in \{\mathsf{Owned}, \mathsf{Unowned}, \mathsf{Shared}\}$
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. Σ, if l is in_{P} Perm then e_1 else $e_2 \to \Sigma, e_2$

By inversion:

1. Perm $\in \{\mathsf{Owned}, \mathsf{Unowned}, \mathsf{Shared}\}$
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-(), $\Gamma; \cdot \vdash_{()} \mathsf{unit} : \cdot \dashv \cdot$. 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) $\Sigma, \boxed{\text{e}}^o \to \Sigma', \boxed{\text{e'}}^o$
2. By inversion:
   (a) $\Gamma; \Delta \vdash_s e : T \dashv \Delta''$
   (b) $\Sigma, e \to \Sigma', e'$
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. $e = l \to_{owned} S(\overline{l'})$

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, l \to_{owned} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()$
   (c) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_l l \to_{owned} S(\overline{x}) : \mathsf{unit} \dashv \Delta^*, l : C\langle\overline{T}\rangle@S$
2. By inversion:
   (a) $\Gamma \vdash T_{ST} <:_* \mathsf{Owned}$
   (b) $\overline{\Gamma; \Delta_0 \vdash_l x : T \dashv \Delta^*}$
   (c) $\overline{\Gamma \vdash T <: type(stateFields(C\langle\overline{T}\rangle, S'))}$
   (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) $\overline{\Gamma \vdash \textit{disposable}\ (T'_{fl})}$
3. Let $\Delta' = \Delta, l : C\langle\overline{T}\rangle@S$. By T-(), $\Gamma; \Delta \vdash_l () : \mathsf{unit} \dashv \Delta$. 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. $e = l \to_{shared} S(\overline{l'})$

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, l \to_{shared} S(\overline{l'}) \to [\mu[\rho(l) \mapsto C\langle\overline{T}\rangle@S(\overline{\rho(l')})]/\mu]\,\Sigma, ()$
2. Now, assume typing rule $\to_{shared}$ applied, since if $\to_{owned}$ applied, then the argument for case E-State-Transition-Static-Ownership (above) applies. Then:
   (a) $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_l l \to_{shared} S(\overline{x}) : \mathsf{unit} \dashv \Delta^*, l : C\langle\overline{T}\rangle@S$

3. By inversion:
   (a) $\Gamma \vdash T_{ST} <:_* \mathsf{Shared}$. By 2, we assume therefore $T_{ST} = \mathsf{Shared}$.
   (b) $\Gamma; \Delta_0 \vdash_l \overline{x} : \overline{T} \dashv \Delta^*$
   (c) $\overline{\Gamma \vdash T <: type(stateFields(C\langle\overline{T}\rangle, S'))}$
   (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) $\overline{\Gamma \vdash \textit{disposable}\ (T'_{fl})}$
   (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) $\Sigma, \text{assert } l \text{ in } T_{ST} \to \Sigma, ()$
2. There are two subcases:
   Subcase: $T_{ST} = \overline{S}$**.** By assumption, $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle.\overline{S} \vdash_s [l@\overline{S'}] : \mathsf{unit} \dashv \Delta_0, l : C\langle\overline{T}\rangle.\overline{S}$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta' \vdash_l () : \mathsf{unit} \dashv \Delta'$. Since $\Sigma' = \Sigma$, $\Delta' = \Delta$, and Γ, Σ, Δ **ok**, we have Γ, Σ', Δ' **ok**.
   Subcase: $T_{ST} \neq \overline{S}$**.** By assumption, $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@T_{ST} \vdash_s [l@T_{ST}] : \mathsf{unit} \dashv \Delta_0, l : C\langle\overline{T}\rangle@T_{ST}$. Let $\Delta' = \Delta$. By T-(), $\Gamma; \Delta' \vdash_l () : \mathsf{unit} \dashv \Delta'$. Since $\Sigma' = \Sigma$, $\Delta' = \Delta$, and Γ, Σ, Δ **ok**, we have Γ, Σ', Δ' **ok**.

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

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, \mathsf{disown}\ l \to \Sigma, l$
2. There are two subcases:
   Subcase: $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle.\overline{S} \vdash_s$ **disown** $l : \mathsf{unit} \dashv \Delta_0, l : T'$**.** By inversion, $C\langle\overline{T}\rangle.\overline{S} \Rrightarrow T/T'$. Let $\Delta' = \Delta''$. By T-(), $\Gamma; \Delta' \vdash_s () : \mathsf{unit} \dashv \Delta'$. 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: $\Gamma; \Delta_0, l : C\langle\overline{T}\rangle@Owned \vdash_s \textsf{disown}\ \ l : \textsf{unit} \dashv \Delta_0, l : T'$**.** By inversion, $C\langle\overline{T}\rangle@Owned \Rrightarrow T/T'$. By T-(), $\Gamma; \Delta' \vdash_l () : \textsf{unit} \dashv \Delta'$. 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. $e = \textsf{pack}$.

1. By assumption, and because e is closed:
   (a) Γ, Σ, Δ **ok**
   (b) $\Sigma, \textsf{pack}\ s \to \Sigma, ()$
   (c) $\Gamma; \Delta_0, l : T, \overline{l.f : T_f} \vdash_l \textsf{pack} : \textsf{unit} \dashv \Delta, l : T$. *(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-(), $\Gamma; \Delta' \vdash_l () : \textsf{unit} \dashv \Delta'$. 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 (A.3.3).

□

Theorem A.3.3 (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* $e = \mathbb{E}[\textsf{disown}\ s]$*, 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), 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. $e =$ let $x : T = e_1$ 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, $\Sigma,$ 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 (A.3.2), 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-$\to_p$. $e = s \to_p S'(\overline{x})$. A rule $E- \to_p$ 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-$\to_p$).

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, $\Sigma, e \rightarrow \Sigma, e_1$ (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 (A.3.21), 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.

□

A.3.1 Supporting Lemmas

Lemma A.3.1 (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 A.3.2 (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 A.3.3 (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 A.3.4 (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 A.3.5 (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 A.3.6 (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 A.3.7 (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 A.3.1 (Non-disposability)**.**

$$\frac{\textit{maybeOwned}\,(T_C@T_{ST}) \qquad \Gamma \vdash isAsset\,(T_C@T_{ST})}{\Gamma \vdash \textit{nonDisposable}\,(T_C@T_{ST})}\ \textsc{nd-owned}$$

Lemma A.3.8 (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 A.3.9 (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 don't have fields, there must not be any field assignments or access involving x, so we don't 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, 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. $T'_{s1} = funcArg(T_C@T_{STs1}, T_C@T_{this}, T_C@T'_{this})$
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. $implementOk_\Gamma\left(I\langle\overline{T}\rangle, M\right)$.

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 A.3.10 (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 A.3.3.1 □

Lemma A.3.11 (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 A.3.12 (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 of the earlier paper [48]. For multiple expressions, simply iterate the argument. □

Lemma A.3.13 (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 A.3.14 (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 $T'_x = T_C@\mathsf{Unowned}$. Split-Unowned applies to T''_x, resulting in $T'''_x = T_C@\mathsf{Unowned}$.
   Case: Split-shared By assumption, $T_x = T_C@\mathsf{Shared}$. If $T''_x = T_C@\mathsf{Shared}$, 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: $T_x = T_C@\mathbf{Owned}$ All subtypes of $T_C@\mathsf{Owned}$ 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 $T_x = T'_x = \mathsf{unit}$. Split-unit applies for T''_x, since the only subtype of unit is unit. Then $T'''_x = \mathsf{unit}$, 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 A.3.3.1 (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 A.3.14 and A.3.13. □

Corollary A.3.3.2 (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 A.3.3.1 and the definition of $\Delta'' <^l_{\Gamma,\Sigma'} \Delta$. □

Lemma A.3.15 (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 (A.3.3). □

Lemma A.3.16 (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 A.3.17 (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 A.3.18 (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 A.3.19 (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 (A.3.20).

Case: Dispose-disposable. $d \notin merge(\Delta^*, \Delta^{**})$, so the conclusion follows immediately from the induction hypothesis.

□

Lemma A.3.20 ($\oplus$ 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 A.3.4 (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 $C\langle\overline{T}\rangle.\overline{S}$, 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 A.3.21 (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 (A.3.8).

□

Appendix B

Java Prescreen (used in user studies for Obsidian)

The prescreening survey was presented to potential participants in the form of a Qualitrics survey. It obtained informed consent, and then asked the following questions:

1. Which of the following might be a valid Java constructor invocation?
   a. `Square.new(5)`
   b. `malloc(sizeof(Square))`
   c. `square(5)`
   d. `new Square(5)`
2. In Java, encapsulation refers to:
   a. Serializing data correctly so that it is transmitted properly between systems
   b. Preventing clients from improperly depending on implementation details
   c. Using the capsule keyword to protect secret data
3. Consider the code below:

```
void test() {
  ArrayList list1 = new ArrayList();
  list1.add(1);
  ArrayList list2 = list1;
  list2.add(2);
  System.out.println(list1.size());
}
```

   If test() is run, what is the output?

   a. 1
   b. 2

4. Which statements are true of interfaces in standard Java?
   a. Methods in interfaces (except for `default` methods) lack bodies.
   b. Methods in interfaces are public by default.

c. A class can implement no more than one interface.

d. Interfaces have no field declarations unless they are `public static final`.

5. Which statements are true of static methods in standard Java?

   a. They are not implemented where they are defined, but must be overridden in a subclass.

   b. They cannot be overridden.

   c. They can be invoked without having an instance of the class.

6. Consider the code below:

```
public abstract class Customer {
  private String name;
  public Customer (String name) {
    this.name = name;
}
  public String getName() { return name; }
  public abstract void buy();
}
```

   Which statements are true about `Customer`?

   a. The `Customer` class cannot be extended.

   b. The `Customer` class cannot be instantiated.

   c. Concrete subclasses must implement the `buy()` method.

   d. Subclasses must implement the `buy()` method.

7. Consider Figure B.1. A software engineer is choosing between designs A and B for a component in which consumers need to find out when certain events occur. Which consideration would best justify choosing Design A over Design B?

   a. The event source should be decoupled from the list of event consumers so that the source need not be aware of the consumers.

   b. None; Design B is better because it is simpler.

   c. Many different sources may need to send the event to different sets of consumers.

   d. Event sources may be implemented by third parties.

8. Again considering the diagram in Figure B.1, what is Design A typically called?

   a. Manage-events

   b. Publish-subscribe

   c. Source-consume

9. A pattern in which there can be only one instance of a given class is called:

   a. Flyweight pattern

   b. Singleton pattern

   c. Unique pattern

   d. Builder pattern

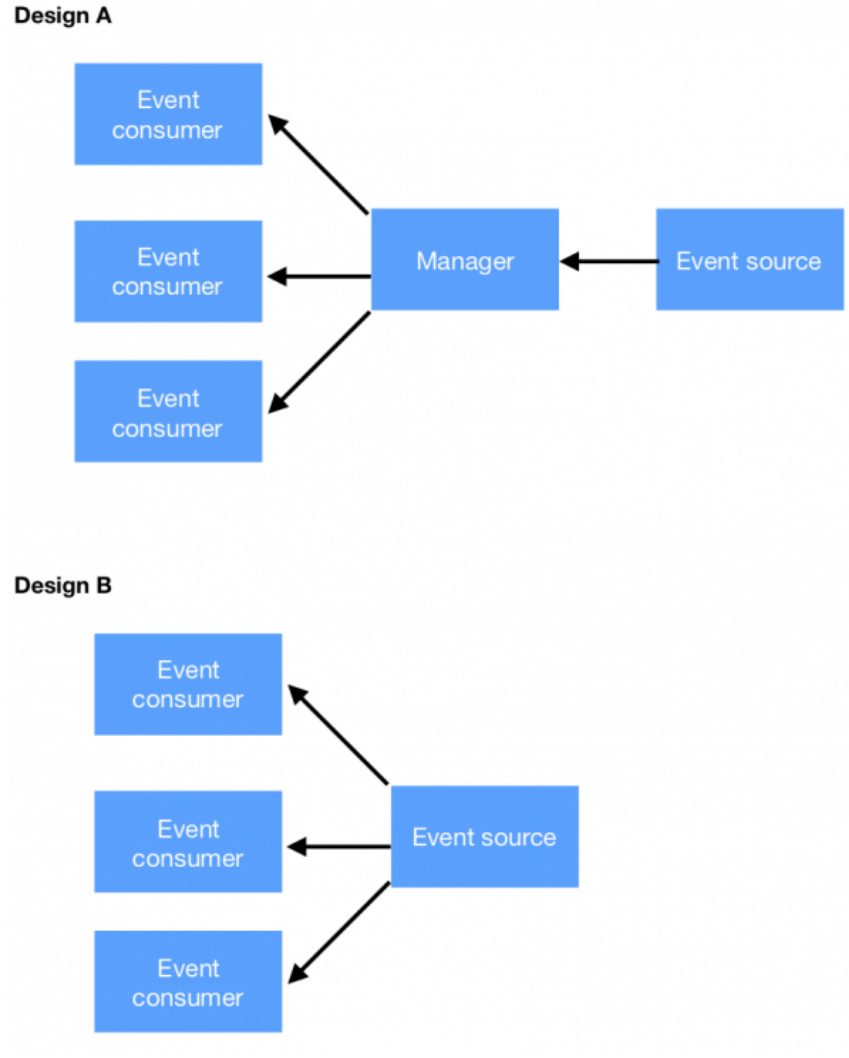

Figure B.1: Two design alternatives.

Appendix C

Case Studies of Obsidian

C.1 Shipping

Obsidian Implementation

Leg.obs

```
1  contract Leg {
2
3    state InTransit;
4    state Arrived {
5      string arrivalDate;
6      string destinationPort;
7    }
8
9    string departureDate;
10   string originPort;
11   Transport@Unowned carrier;
12
13   Leg@InTransit(Transport@Unowned t, string ddepartureDate, string doriginPort) {
14     ->InTransit(departureDate = ddepartureDate, originPort = doriginPort, carrier = t);
15   }
16
17   transaction getDepartureDate(Leg@Owned this) returns string {
18     return departureDate;
19   }
20
21   transaction getOriginPort(Leg@Owned this) returns string {
22     return originPort;
23   }
24
25   transaction getCarrier(Leg@Owned this) returns Transport@Unowned {
26     return carrier;
27   }
28
29
30   transaction setArrival(Leg@InTransit >> Arrived this, string date, string port) {
31     ->Arrived(arrivalDate = date, destinationPort = port);
32   }
33 }
```

Shipment.obs

```
1  import "IO.obs"
2  import "Transport.obs"
3  import "LinkedList.obs"
4
```

```
5
6  main contract ShippingDriver {
7
8    int totalShipments;
9    int nextID;
10
11   ShippingDriver@Owned() {
12     totalShipments = 0;
13     nextID = 1;
14   }
15
16   transaction createAgreement(string pseller,
17                               string pshipper,
18                               string pbuyer,
19                               string psource,
20                               string pdest,
21                               int pload,
22                               string pdescription,
23                               string pplannedDate) returns Shipment@Contract {
24     Shipment s = new Shipment(nextID,
25                               pseller,
26                               pshipper,
27                               pbuyer,
28                               psource,
29                               pdest,
30                               pload,
31                               pdescription,
32                               pplannedDate);
33     totalShipments = totalShipments + 1;
34     nextID = nextID + 1;
35     return s;
36   }
37 }
38
39
40 contract Shipment {
41   int id;
42   string seller;
43   string shipper;
44   string buyer;
45   string source;
46   string dest;
47   int load;
48   string description;
49   string plannedDate;
50
51   LinkedList@Owned transportList;
52
53   state Contract;
54   state Release {
55     string releasedDate;
56   }
57   state Sailing;
58   state InTransit {
59     Transport@InTransport inTsp;
60   }
61   state Delivered;
62
63   Shipment@Contract(int nid,
64                     string pseller,
65                     string pshipper,
66                     string pbuyer,
67                     string psource,
68                     string pdest,
69                     int pload,
70                     string pdescription,
```

```
71                string pplannedDate) {
72      id = nid;
73      seller = pseller;
74      shipper = pshipper;
75      buyer = pbuyer;
76      seller = pseller;
77      source = psource;
78      dest = pdest;
79      load = pload;
80      description = pdescription;
81      plannedDate = pplannedDate;
82      transportList = new LinkedList[Transport@Owned]();
83      ->Contract;
84
85      IO io = new IO();
86      io.println("");
87      io.print("* ");
88      io.print(pplannedDate);
89      io.print(" - shipment ");
90      io.printInt(id);
91      io.println(" [Contract]");
92      io.print("         seller: ");
93      io.println(seller);
94      io.print("         buyer: ");
95      io.println(buyer);
96      io.print("         shipper: ");
97      io.println(shipper);
98      io.print("         source port: ");
99      io.println(source);
100     io.print("         destination port: ");
101     io.println(dest);
102     io.print("         shipment description: ");
103     io.println(description);
104     io.println("");
105   }
106
107
108   transaction release(Shipment@Contract >> Release this, string date) {
109     ->Release(releasedDate = date);
110
111     IO io = new IO();
112     io.println("");
113     io.print("* ");
114     io.print(date);
115     io.print(" - shipment ");
116     io.printInt(id);
117     io.println(" [Release]");
118     io.println("");
119   }
120
121
122   transaction setSail(Shipment@Release >> Sailing this, string date) {
123     ->Sailing;
124
125     IO io = new IO();
126     io.println("");
127     io.print("* ");
128     io.print(date);
129     io.print(" - shipment ");
130     io.printInt(id);
131     io.println(" [InTransit]");
132     io.println("");
133   }
134
135
136   transaction depart(Shipment@Sailing >> InTransit this,
```

```
137                       string transportName,
138                       string transportType,
139                       string date,
140                       string port) {
141       Transport newTransport = new Transport(transportName, transportType, source, date);
142       [newTransport @ Load];
143       newTransport.depart(date, port);
144       [newTransport @ InTransport];
145       ->InTransit(inTsp = newTransport);
146     }
147
148
149     transaction layover(Shipment@InTransit this, string date, string port) {
150       inTsp.stopOver(date, port);
151     }
152
153
154     transaction layoverDepart(Shipment@InTransit this, string date, string port) {
155       inTsp.layoverDepart(date, port);
156     }
157
158
159     transaction transfer(Shipment@InTransit this,
160                          string transportName,
161                          string transportType,
162                          string date,
163                          string port) {
164       inTsp.arrive(date, port);
165       transportList.append[Transport](inTsp);
166
167       Transport newTransport = new Transport(transportName, transportType, source, date);
168       [newTransport @ Load];
169       newTransport.depart(date, port);
170       [newTransport @ InTransport];
171       inTsp = newTransport;
172
173       IO io = new IO();
174       io.print("* ");
175       io.print(date);
176       io.print(" - transfer at port [");
177       io.print(port);
178       io.println("]");
179       io.println("");
180     }
181
182     transaction deliver(Shipment@InTransit >> Delivered this, string date, string port) {
183       inTsp.arrive(date, port);
184       transportList.append[Transport](inTsp);
185       ->Delivered;
186
187       IO io = new IO();
188       io.print("* ");
189       io.print(date);
190       io.print(" - shipment ");
191       io.printInt(id);
192       io.print(" [Delivered] at port [");
193       io.print(port);
194       io.println("]");
195       io.println("");
196     }
197
198     transaction getID(Shipment this) returns int {
199       return id;
200     }
201
202   }
```

ShipmentClient.obs

```
import "Shipment.obs"

main contract ShippingDriverClient {

  transaction main(remote ShippingDriver@Shared sh) {

    remote Shipment s = sh.createAgreement("Dole",
               "TruckMyShipment",
               "ShopRite",
               "Sunnyvale, California",
               "Bronx, New York",
               50,
               "Strawberries",
               "12/01/2018");
    s.release("12/02/2018");
    s.setSail("12/02/2018");

    // transport 1
    s.depart("Truck5000", "truck", "12/03/2018", "Sunnyvale, California");
    s.layover("12/06/2018", "Salt Lake City, Utah");
    s.layoverDepart("12/06/2018", "Salt Lake City, Utah");

    // transport 2
    s.transfer("Fedex", "Cargo", "12/07/2018", "Chicago, Illinois");
    s.deliver("12/09/2018", "Bronx, New York");
   }
}
```

Transport.obs

```
import "Leg.obs"
import "LinkedList.obs"

contract Transport {

  state Load;
  state InTransport {
    Leg@InTransit currentLeg;
  }
  state Unload {
    string destinationPort;
  };


  string transportName;
  string transportType;
  string originPort;
  LinkedList[Leg@Owned]@Owned legList;


  Transport@Load(string pTransportName, string pTransportType, string pOriginPort, string date) {
    transportName = pTransportName;
    transportType = pTransportType;
    originPort = pOriginPort;
    legList = new LinkedList[Leg@Owned]();
    ->Load;

    IO io = new IO();
    io.println("");
    io.print("* ");
    io.print(date);
    io.print(" - transport [Load] on [");
    io.print(pTransportName);
    io.print("], type [");
    io.print(pTransportType);
```

```
36      io.print("] at port [");
37      io.print(pOriginPort);
38      io.println("]");
39    }
40
41
42    transaction getTransportName(Transport this) returns string {
43      return transportName;
44    }
45
46
47    transaction getTransportType(Transport this) returns string {
48      return transportType;
49    }
50
51
52    transaction getOriginPort(Transport this) returns string {
53      return originPort;
54    }
55
56    transaction getDestinationPort(Transport@Unload this) returns string {
57      return destinationPort;
58    }
59
60    transaction depart(Transport@Load >> InTransport this, string pdepartureDate, string port) {
61      Leg newLeg = new Leg(this, pdepartureDate, port);
62      [newLeg @ InTransit];
63      ->InTransport(currentLeg = newLeg);
64
65      IO io = new IO();
66      io.println("");
67      io.print("* ");
68      io.print(pdepartureDate);
69      io.print(" - leg [InTransit]. port [");
70      io.print(port);
71      io.println("]");
72      io.print("* ");
73      io.print(pdepartureDate);
74      io.print(" - transport [InTransport]. port [");
75      io.print(port);
76      io.println("]");
77
78    }
79
80    transaction stopOver(Transport@InTransport this, string parrivalDate, string port) {
81      currentLeg.setArrival(parrivalDate, port);
82      [currentLeg @ Arrived];
83      legList.append(currentLeg);
84      [currentLeg @ Unowned];
85      currentLeg = new Leg(this, "", "");
86
87      IO io = new IO();
88      io.println("");
89      io.print("* ");
90      io.print(parrivalDate);
91      io.print(" - leg [Arrived]. port [");
92      io.print(port);
93      io.println("]");
94      io.println("");
95    }
96
97
98    transaction layoverDepart(Transport@ InTransport this, string date, string port) {
99      Leg newLeg = new Leg(this, date, port);
100     ->InTransport(currentLeg = newLeg);
101
```

```
    IO io = new IO();
    io.println("");
    io.print("* ");
    io.print(date);
    io.print(" - leg [InTransit]. port [");
    io.print(port);
    io.println("]");
    io.println("");
  }


  transaction arrive(Transport@InTransport >> Unload this, string date, string port) {
    currentLeg.setArrival(date, port);
    [currentLeg @ Arrived];
    legList.append(currentLeg);
    [currentLeg @ Unowned];
    ->Unload(destinationPort = port);

    IO io = new IO();
    io.println("");
    io.print("* ");
    io.print(date);
    io.print(" - leg [Arrived]. port [");
    io.print(port);
    io.println("]");

    io.print("* ");
    io.print(date);
    io.print(" - transport [Unload]. port [");
    io.print(port);
    io.println("]");
  }

}
```

Solidity Implementation

Migrations.sol

```
pragma solidity ^0.5.8;

contract Migrations {
  address public owner;

  // A function with the signature `last_completed_migration()`, returning a uint, is required.
  uint public last_completed_migration;

  modifier restricted() {
    if (msg.sender == owner) _;
  }

  constructor () public {
    owner = msg.sender;
  }

  // A function with the signature `setCompleted(uint)` is required.
  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }

  function upgrade(address new_address) public restricted {
    Migrations upgraded = Migrations(new_address);
    upgraded.setCompleted(last_completed_migration);
  }
}
```

Shipment.sol

```
pragma solidity ^0.5.8;

import "./Transport.sol";
import "./Utils.sol";

contract ShippingDriver {
    int totalShipments;
    int nextID;

    constructor() public {
        totalShipments = 0;
        nextID = 1;
    }

    event NewShipment(Shipment);

    function createAgreement(string memory pseller,
                             string memory pshipper,
                             string memory pbuyer,
                             string memory psource,
                             string memory pdest,
                             int pload,
                             string memory pdescription,
                             string memory pplannedDate)
                                public returns (Shipment) {
        Shipment s = new Shipment(nextID,
                                  pseller,
                                  pshipper,
                                  pbuyer,
                                  psource,
                                  pdest,
                                  pload,
                                  pdescription);
        s.setPlannedDate(pplannedDate);
        totalShipments = totalShipments + 1;
        nextID = nextID + 1;

        emit NewShipment(s);
        return s;
    }
}

contract Shipment {
    enum ShipmentState {Contract, Release, Sailing, InTransit, Delivered}

    ShipmentState public state;

    int id;
    string seller;
    string shipper;
    string buyer;
    string source;
    string dest;
    int load;
    string description;
    string plannedDate;
    string releasedDate;
    Transport inTsp;

    TransportList transportList;

    event DeliveryComplete(string date);

    constructor (int nid,
                 string memory pseller,
```

```
66                  string memory pshipper,
67                  string memory pbuyer,
68                  string memory psource,
69                  string memory pdest,
70                  int pload,
71                  string memory pdescription) public {
72          id = nid;
73          seller = pseller;
74          shipper = pshipper;
75          buyer = pbuyer;
76          seller = pseller;
77          source = psource;
78          dest = pdest;
79          load = pload;
80          description = pdescription;
81          transportList = new TransportList();
82          state = ShipmentState.Contract;
83      }
84
85      function setPlannedDate (string memory pplannedDate) public {
86          plannedDate = pplannedDate;
87      }
88
89
90      function release(string memory date) public {
91          assert(state == ShipmentState.Contract);
92          releasedDate = date;
93          state = ShipmentState.Release;
94      }
95
96
97      function setSail() public {
98          assert(state == ShipmentState.Release);
99          state = ShipmentState.Sailing;
100     }
101
102
103     function depart(string memory transportName,
104                     string memory transportType,
105                     string memory date,
106                     string memory port) public {
107         assert(state == ShipmentState.Sailing);
108         Transport newTransport = new Transport(transportName, transportType, source);
109         newTransport.depart(date, port);
110         inTsp = newTransport;
111         state = ShipmentState.InTransit;
112     }
113
114
115     function layover(string memory date, string memory port) public {
116         assert(state == ShipmentState.InTransit);
117         inTsp.stopOver(date, port);
118     }
119
120     function layoverDepart(string memory date, string memory port) public {
121         assert(state == ShipmentState.InTransit);
122
123         inTsp.layoverDepart(date, port);
124     }
125
126
127     function transfer(string memory transportName,
128                       string memory transportType,
129                       string memory date,
130                       string memory port) public {
131         assert(state == ShipmentState.InTransit);
```

```
132
133         inTsp.arrive(date, port);
134         transportList.append(inTsp);
135
136         Transport newTransport = new Transport(transportName, transportType, source);
137         newTransport.depart(date, port);
138         inTsp = newTransport;
139     }
140
141     function deliver(string memory date, string memory port) public {
142         assert(state == ShipmentState.InTransit);
143
144         inTsp.arrive(date, port);
145         transportList.append(inTsp);
146         state = ShipmentState.Delivered;
147
148         emit DeliveryComplete(date);
149     }
150
151     function getID() public view returns (int) {
152         return id;
153     }
154
155 }
```

Transport.sol

```
1  pragma solidity ^0.5.8;
2
3  contract Transport {
4      enum LoadState {Load, InTransport, Unload}
5
6      LoadState state;
7
8      Leg currentLeg;
9      string destinationPort;
10     string transportName;
11     string transportType;
12     string originPort;
13     LegList legList;
14
15     constructor (string memory pTransportName,
16                  string memory pTransportType,
17                  string memory pOriginPort) public {
18         transportName = pTransportName;
19         transportType = pTransportType;
20         originPort = pOriginPort;
21         state = LoadState.Load;
22
23         legList = new LegList();
24     }
25
26     function depart(string memory pdepartureDate, string memory port) public {
27         assert(state == LoadState.Load);
28         Leg newLeg = new Leg(this, pdepartureDate, port);
29         currentLeg = newLeg;
30
31         state = LoadState.InTransport;
32     }
33
34     function stopOver(string memory parrivalDate, string memory port) public {
35         assert(state == LoadState.InTransport);
36         currentLeg.setArrival(parrivalDate, port);
37         legList.append(currentLeg);
38         currentLeg = new Leg(this, "", "");
39     }
40
```

```
41      function layoverDepart(string memory date, string memory port) public {
42          assert(state == LoadState.InTransport);
43          currentLeg = new Leg(this, date, port);
44      }
45
46      function arrive(string memory date, string memory port) public {
47          assert(state == LoadState.InTransport);
48          currentLeg.setArrival(date, port);
49          legList.append(currentLeg);
50          destinationPort = port;
51          state = LoadState.Unload;
52      }
53  }
54
55  contract TransportList {
56      Transport[] transports;
57
58      constructor() public {
59          transports = new Transport[](0);
60      }
61
62      function append (Transport pTransport) public {
63          transports.push(pTransport);
64      }
65
66      function length () public view returns (uint) {
67          return transports.length;
68      }
69
70      function nth (uint n) public view returns (Transport) {
71          assert (n < transports.length);
72          return transports[n];
73      }
74  }
75
76  contract Leg {
77      enum LegState {InTransit, Arrived}
78
79      LegState state;
80      string arrivalDate;
81      string destinationPort;
82      string departureDate;
83      string originPort;
84      Transport carrier;
85
86      constructor (Transport t, string memory ddepartureDate, string memory doriginPort) public {
87          departureDate = ddepartureDate;
88          originPort = doriginPort;
89          carrier = t;
90
91          state = LegState.InTransit;
92      }
93
94      function getDepartureDate() public view returns (string memory s)  {
95          return departureDate;
96      }
97
98      function setArrival(string memory date, string memory port) public {
99          assert(state == LegState.InTransit);
100         state = LegState.Arrived;
101
102         arrivalDate = date;
103         destinationPort = port;
104     }
105 }
106
```

```
contract LegList {
    Leg[] legs;

    constructor() public {
        legs = new Leg[](0);
    }

    function append (Leg pLeg) public {
        legs.push(pLeg);
    }

    function length () public view returns (uint) {
        return legs.length;
    }

    function nth (uint n) public view returns (Leg) {
        assert (n < legs.length);
        return legs[n];
    }
}
```

Utils.sol

```
pragma solidity ^0.5.8;

contract Utils {
    function equalStrings(string memory a, string memory b) public pure returns (bool) {
        if(bytes(a).length != bytes(b).length) {
            return false;
        } else {
            return keccak256(abi.encodePacked(a)) == keccak256(abi.encodePacked(b));
        }
    }
}
```

C.2 Insurance

Obsidian Implementation

ActivePolicyMap.obs

```
import "PolicyRecord.obs"


// This implements a mutable linked list of Active policies.
contract ActivePolicyList {
  asset state HasNext {
    ActivePolicyList@Owned next;
  }

  asset state NoNext;
  state Empty;

  Policy@Unowned key available in HasNext, NoNext;
  PolicyRecord@Active val available in HasNext, NoNext;


  ActivePolicyList@Empty() {
    ->Empty;
  }

  transaction getKey(ActivePolicyList@HasNext this) returns Policy@Unowned {
    return key;
```

```
23    }
24
25    transaction getValue(ActivePolicyList@HasNext this) returns PolicyRecord@Unowned {
26      return val;
27    }
28
29    transaction disconnectNext(ActivePolicyList@HasNext >> NoNext this)
30                returns ActivePolicyList@Owned {
31      ActivePolicyList removedNext = next;
32      ->NoNext;
33      return removedNext;
34    }
35
36    transaction reconnectNext(ActivePolicyList@NoNext >> HasNext this,
37                              ActivePolicyList@Owned >> Unowned newNext) {
38      ->HasNext(next = newNext);
39    }
40
41    private transaction removeValue(ActivePolicyList@NoNext >> Empty this)
42              returns PolicyRecord@Active {
43      PolicyRecord valCopy = val;
44      ->Empty;
45      return valCopy;
46    }
47
48
49    transaction append(ActivePolicyList@Owned this, Policy@Unowned k,
50                       PolicyRecord@Active >> Unowned v) {
51      switch this {
52        case HasNext {
53          next.append(k, v);
54        }
55        case Empty {
56          ->HasNext(key = k, val = v, next = new ActivePolicyList());
57        }
58      }
59    }
60
61
62
63    // Removes prevNode.next, returning the PolicyRecord that was previously owned by prevNode.next.
64    // Precondition: this is not the last node of the linked list.
65    transaction removeNext(ActivePolicyList@HasNext this) returns PolicyRecord@Active {
66      ActivePolicyList removedNode = this.next;
67      [removedNode@Owned];
68
69      switch removedNode {
70        case Empty {
71          revert; // this case doesn't make any sense.
72        }
73        case HasNext {
74          this.next = removedNode.disconnectNext();
75          [removedNode@NoNext];
76          return removedNode.removeValue();
77        }
78      }
79    }
80
81    // Assumes "this" is not a match.
82     transaction removeInRest(ActivePolicyList@Owned this, Policy@Unowned policy)
83                 returns PolicyRecord@Active {
84        switch this {
85           case Empty {
86             revert; // We didn't find the policy in the list.
87           }
88           case HasNext {
```

```
            switch next {
             case Empty {
               revert;
             }
             case HasNext {
               if(next.getKey().equals(policy)) {
                 return this.removeNext();
               }
               else {
                 return next.removeInRest(policy);
               }
             }
            }
          }
      }
  }
}


main asset contract ActivePolicyMap {
  ActivePolicyList@Owned list;

  ActivePolicyMap@Owned() {
    list = new ActivePolicyList(); // Start with an empty list.
  }

  transaction set(ActivePolicyMap@Owned this,
                  Policy@Unowned key,
                  PolicyRecord@Active >> Unowned val) {
    list.append(key, val);
  }




  // remove and return value of the pair with the given policy as a key
  transaction get(ActivePolicyMap@Owned this,
                  Policy@Unowned policy) returns PolicyRecord@Active {
    PolicyRecord removed;

    switch list {
      case Empty {
        revert; // Element not in map.
      }
      case HasNext {
        if(list.getKey().equals(policy)) {
          ActivePolicyList secondNode = list.disconnectNext();
          [list@NoNext];
          removed = list.removeValue();
          list = secondNode;
        } else {
          removed = list.removeInRest(policy);
        }
      }
    }

    return removed;

  }


}
```

Bank.obs

```
import "Money.obs"

```

```
main asset contract Bank {
  Money@Owned mon;
  // the bank's private key for making tokens
  int key;

  Bank@Owned() {
    // TODO: generate key
    mon = new Money(1000000);
    key = 35;
  }

  transaction withdrawMoney(int amount) returns Money@Owned{
     return mon.getAmountOfMoney(amount);
  }

  transaction depositMoney(Money@Owned >> Unowned m) {
    mon.addMoney(m);
  }

}
```

InsuranceBid.obs

```
import "Money.obs"

main contract InsuranceBid {
  int cost;
  int expirationTime;

  asset state HoldingPayout {
    Money@Owned payout;
  }
  state NoPayout;

  InsuranceBid@HoldingPayout(int c, int expiration, Money@Owned >> Unowned m) {
    cost = c;
    expirationTime = expiration;
    ->HoldingPayout(payout = m);
  }

  transaction getCost(InsuranceBid@Unowned this) returns int {
    return cost;
  }

  transaction getExpirationTime(InsuranceBid@Unowned this) returns int {
    return expirationTime;
  }

  transaction getPayout(InsuranceBid@HoldingPayout >> NoPayout this) returns Money@Owned {
    Money p = payout;
    ->NoPayout;
    return p;
  }


}
```

InsuranceClient.obs

```
import "InsuranceService.obs"
import "IO.obs"

main contract InsuranceClient {
  transaction main(remote InsuranceService@Shared insService) {
    TimeService@Shared timeService = new TimeService();
    Bank@Shared bank = new Bank();
    // should be requesting money from bank, but can't call transactions from constructor
```

```
    Money@Owned money = new Money(25);
    Insurer@Owned insurers = new Insurer(timeService, bank, money);

    insService.addInsurer(insurers);

    Policy@Offered policy = insService.requestBids(3);

    // test
    new IO().println(policy.getCost());

    // now buy the policy
    Money@Owned refund = insService.buyPolicy(policy, new Money(10));

    new IO().println(refund.getAmount());

  }
}
```

InsuranceService.obs

```
import "Insurer.obs"
import "WeatherRecord.obs"
import "Signature.obs"
import "PolicyRecord.obs"
import "PendingPolicyMap.obs"
import "ActivePolicyMap.obs"


main asset contract InsuranceService {
  // TODO: Collection of Insurer
  // TODO: Keys of trusted banks
  state Ready {
    Insurer@Shared insurer;
    TimeService@Shared timeService;
    // for now have to have a reference to bank
    Bank@Shared bank;
  }


  state New {
  }

  // pending and active policies
  PendingPolicyMap@Owned pendingPolicies;
  ActivePolicyMap@Owned activePolicies;


  InsuranceService@Owned() {
    pendingPolicies = new PendingPolicyMap();
    activePolicies = new ActivePolicyMap();
    ->New();
  }

  transaction setup(InsuranceService@New >> Ready this,
                    TimeService@Shared ts,
                    Bank@Shared b,
                    Insurer@Shared ins) {
    ->Ready(insurer = ins, timeService = ts, bank = b);
  }

  transaction testSetup(InsuranceService@New >> Ready this) {
    // Convenience transaction for testing.
    TimeService ts = new TimeService();
    Bank b = new Bank();
    Money m = b.withdrawMoney(10000);
    Insurer ins = new Insurer(ts, b, m);
    ->Ready(insurer = ins, timeService = ts, bank = b);
```

```
48    }
49
50    transaction getTimeService(InsuranceService@Ready this) returns TimeService@Shared {
51      return timeService;
52    }
53
54    transaction getBank(InsuranceService@Ready this) returns Bank@Shared {
55      return bank;
56    }
57
58    // All bids (for now) are for one-month periods. They insure within
59    // "radius" of the given coordinate:
60    // if the moisture content ever drops below the threshold, then the policy pays out.
61    transaction requestBid(InsuranceService@Ready this,
62                           int longitude,
63                           int latitude,
64                           int radius,
65                           int moistureContent,
66                           int payout) returns Policy@Offered {
67      InsuranceBid insBid = insurer.requestBid(longitude, latitude, radius, moistureContent, payout);
68
69      int cost = insBid.getCost();
70      int expiration = insBid.getExpirationTime();
71      Money m = insBid.getPayout();
72
73      Policy policy = new Policy(cost, expiration, longitude, latitude, radius, moistureContent);
74      PolicyRecord pendingPolicy = new PolicyRecord(policy, m);
75
76      pendingPolicies.set(policy, pendingPolicy);
77
78      return policy;
79    }
80
81    transaction buyPolicy(InsuranceService@Ready this,
82                          Policy@Offered >> (Active | Expired) policy,
83                          Money@Owned >> Unowned money) returns Money@Owned {
84
85      if (timeService.getTime() > policy.getExpirationTime()) {
86        PolicyRecord pendingPolicy = pendingPolicies.get(policy); [pendingPolicy @ Pending];
87
88        // refund insurer's payout
89        Money insurerRefund = pendingPolicy.refund();
90        [pendingPolicy @ Expired];
91
92        insurer.receiveRefund(insurerRefund);
93
94        // expire policy and return money
95        disown pendingPolicy;
96        policy.expire();
97        return money;
98      } else {
99        PolicyRecord pendingPolicy = pendingPolicies.get(policy);
100       // get payment
101       int cost = policy.getCost();
102       Money payment = money.getAmountOfMoney(cost);
103
104       // activate policy record with payment
105       pendingPolicy.activate(payment);
106       activePolicies.set(policy, pendingPolicy);
107
108       // activate policy and return any change
109       policy.activate();
110       return money;
111     }
112   }
113
```

```
114   // Returns a payment.
115   transaction claim(InsuranceService@Ready this,
116                     Policy@Active >> Claimed policy,
117                     WeatherRecord@Shared weatherRecord,
118       Signature s) returns Money@Owned {
119     if (!s.verify(weatherRecord)) {
120       // Invalid signature; can't pay a claim with it.
121       revert "Unable to verify weather record signature.";
122     }
123
124     if (policy.isClaimableWithWeather(weatherRecord)) {
125
126       // get will revert if the policy isn't found.
127       PolicyRecord record = activePolicies.get(policy);
128       Money payment = record.claim();
129
130       policy.claim();
131
132       return payment;
133     }
134     else {
135       revert "The policy cannot be claimed because it does not apply to the given weather record.";
136     }
137   }
138
139 }
```

Insurer.obs

```
1  import "InsuranceBid.obs"
2  import "Bank.obs"
3  import "TimeService.obs"
4
5  // TODO : Collections
6
7  main asset contract Insurer {
8    Money@Owned money;
9    Bank@Unowned bank;
10   TimeService@Shared timeService;
11
12   Insurer@Owned(TimeService@Shared ts, Bank@Unowned b, Money@Owned >> Unowned m) {
13     timeService = ts;
14     bank = b;
15     money = m;
16   }
17
18   // TODO : Calculate Bid using some needed information, for now just int
19   transaction requestBid(int longitude,
20           int latitude,
21           int radius,
22           int moistureContent,
23           int payout) returns InsuranceBid@HoldingPayout {
24     // For now, assume 10% of policies will have to pay out, and we're keeping a $10 profit.
25     // Obviously this is oversimplified.
26     int costOfBid = payout / 10 + 10;
27
28
29     // available for 24 hours
30     int twentyFourHours = timeService.hoursToMillis(24);
31     int expirationTime = timeService.getTime() + twentyFourHours;
32
33     int insurerEscrowAmount = payout - costOfBid;
34
35     if (money.getAmount() < insurerEscrowAmount) {
36       revert "Insurer does not have sufficient funds to offer another bid.";
37     }
38
```

```
    // get correct amount of money from owned money
    Money escrowMoney = money.getAmountOfMoney(insurerEscrowAmount);

    InsuranceBid b = new InsuranceBid(costOfBid, expirationTime, escrowMoney);

    return b;
  }

  transaction receiveRefund(Money@Owned >> Unowned m) {
    money.addMoney(m);
  }

  transaction getRemainingBalance() returns int {
    return money.getAmount();
  }

}
```

Money.obs

```
asset contract Money {
    int amount;

    Money@Owned(int amt) {
        // TODO: confirm that bank can only make money
        if (1 + 2 == 3) {
            amount = amt;
        } else {
            revert;
        }
    }

    transaction addMoney(Money this, Money@Owned >> Unowned m) {
         amount = amount + m.getAmount();
         disown m;
    }

    transaction getAmount(Money this) returns int {
        return amount;
    }

    transaction getAmountOfMoney(Money this, int amt) returns Money@Owned {
        if (amt > amount) {
          // TODO: or should this return as much as they have (amount)
            revert "Can't split out more money than is available in a given Money object.";
        } else {
            amount = amount - amt;
            return new Money(amt);
        }
    }

}
```

PendingPolicyMap.obs

```
import "PolicyRecord.obs"


// This implements a mutable linked list of Pending policies.
contract PendingPolicyList {
  asset state HasNext {
    PendingPolicyList@Owned next;
  }

  asset state NoNext;
  state Empty;

```

```
13  Policy@Unowned key available in HasNext, NoNext;
14  PolicyRecord@Pending val available in HasNext, NoNext;
15
16
17  PendingPolicyList@Empty() {
18    ->Empty;
19  }
20
21  transaction getKey(PendingPolicyList@HasNext this) returns Policy@Unowned {
22    return key;
23  }
24
25  transaction getValue(PendingPolicyList@HasNext this) returns PolicyRecord@Unowned {
26    return val;
27  }
28
29  transaction disconnectNext(PendingPolicyList@HasNext >> NoNext this)
30                             returns PendingPolicyList@Owned {
31    PendingPolicyList removedNext = next;
32    ->NoNext;
33    return removedNext;
34  }
35
36  transaction reconnectNext(PendingPolicyList@NoNext >> HasNext this,
37                            PendingPolicyList@Owned >> Unowned newNext) {
38    ->HasNext(next = newNext);
39  }
40
41  private transaction removeValue(PendingPolicyList@NoNext >> Empty this)
42                      returns PolicyRecord@Pending {
43    PolicyRecord valCopy = val;
44    ->Empty;
45    return valCopy;
46  }
47
48
49  transaction append(PendingPolicyList@Owned this,
50                     Policy@Unowned k,
51                     PolicyRecord@Pending >> Unowned v) {
52    switch this {
53      case HasNext {
54        next.append(k, v);
55      }
56      case Empty {
57        ->HasNext(key = k, val = v, next = new PendingPolicyList());
58      }
59    }
60  }
61
62
63
64  // Removes prevNode.next, returning the PolicyRecord that was previously owned by prevNode.next.
65  // Precondition: this is not the last node of the linked list.
66  transaction removeNext(PendingPolicyList@HasNext this) returns PolicyRecord@Pending {
67    PendingPolicyList removedNode = this.next;
68    [removedNode@Owned];
69
70    switch removedNode {
71      case Empty {
72        revert; // this case doesn't make any sense.
73      }
74      case HasNext {
75        this.next = removedNode.disconnectNext();
76        [removedNode@NoNext];
77        return removedNode.removeValue();
78      }
```

```
79      }
80    }
81
82    // Assumes "this" is not a match.
83     transaction removeInRest(PendingPolicyList@Owned this,
84                              Policy@Unowned policy) returns PolicyRecord@Pending {
85        switch this {
86           case Empty {
87             revert; // We didn't find the policy in the list.
88           }
89           case HasNext {
90             switch next {
91              case Empty {
92                revert;
93              }
94              case HasNext {
95                if(next.getKey().equals(policy)) {
96                  return this.removeNext();
97                }
98                else {
99                  return next.removeInRest(policy);
100               }
101             }
102           }
103        }
104       }
105   }
106 }
107
108
109 main asset contract PendingPolicyMap {
110   PendingPolicyList@Owned list;
111
112   PendingPolicyMap@Owned() {
113     list = new PendingPolicyList(); // Start with an empty list.
114   }
115
116   transaction set(PendingPolicyMap@Owned this,
117           Policy@Unowned key,
118           PolicyRecord@Pending >> Unowned val) {
119     list.append(key, val);
120   }
121
122
123
124
125   // remove and return value of the pair with the given policy as a key
126   transaction get(PendingPolicyMap@Owned this,
127                   Policy@Unowned policy) returns PolicyRecord@Pending {
128     PolicyRecord removed;
129
130     switch list {
131       case Empty {
132         revert; // Element not in map.
133       }
134       case HasNext {
135         if(list.getKey().equals(policy)) {
136           PendingPolicyList secondNode = list.disconnectNext();
137           [list@NoNext];
138           removed = list.removeValue();
139           list = secondNode;
140         } else {
141           removed = list.removeInRest(policy);
142         }
143       }
144     }
```

```
    return removed;

  }
}
```

PolicyRecord.obs

```
import "Policy.obs"
import "Money.obs"


main contract PolicyRecord {
  Policy@Unowned policy;

  asset state Pending {
    Money@Owned payout;
  }

  asset state Active {
    Money@Owned claim;
  }

  state Expired {}
  state Claimed {}

  PolicyRecord@Pending(Policy@Unowned p, Money@Owned >> Unowned m) {
    policy = p;
    Pending::payout = m;
    ->Pending;
  }

  transaction getPolicy(PolicyRecord this) returns Policy@Unowned {
    return policy;
  }

  transaction refund(PolicyRecord@Pending >> Expired this) returns Money@Owned {
     Money refund = payout;
     ->Expired;
     return refund;
  }

  transaction activate(PolicyRecord@Pending >> Active this, Money@Owned >> Unowned policyCost) {
     payout.addMoney(policyCost);
     Active::claim = payout;
     ->Active;
  }

  transaction claim(PolicyRecord@Active >> Claimed this) returns Money@Owned {
    Money m = claim;
    ->Claimed;
    return m;
  }

}
```

Signature.obs

```
import "WeatherRecord.obs"

contract Signature {
  string signature; // If we had arrays of bytes, we'd use those, but we don't.

  // Constructor makes a Signature object that corresponds to a particular object.
  Signature@Owned (WeatherRecord c) {
      signature = ""; // TODO
  }
```

```
10
11    // verify takes a contract and returns true iff the receiver is a signature of that contract.
12    transaction verify(Signature@Unowned this, WeatherRecord c) returns bool {
13        return true; // TODO
14    }
15  }
```

TimeService.obs

```
1   main contract TimeService {
2     int currentTimeInMillis;
3
4     TimeService@Shared() {
5       currentTimeInMillis = 0;
6     }
7
8     transaction getTime() returns int {
9       return currentTimeInMillis;
10    }
11
12    transaction incrTime() {
13      currentTimeInMillis = currentTimeInMillis + 1;
14    }
15
16    transaction hoursToMillis(int hours) returns int {
17      return hours * 3600000;
18    }
19
20    transaction setTime(int newMillis) {
21      if (newMillis >= currentTimeInMillis) {
22        currentTimeInMillis = newMillis;
23      }
24      else {
25        revert "Can't move time backwards.";
26      }
27    }
28  }
```

WeatherRecord.obs

```
1   // A WeatherRecord represents a report from an external weather service.
2   // It records a soil moisture measurement at a particular location and time.
3
4   contract WeatherRecord {
5       int longitude;
6       int latitude;
7       int moisture;
8       int timestamp;
9
10      WeatherRecord@Shared(int lo, int la, int m, int t) {
11          longitude = lo;
12          latitude = la;
13          moisture = m;
14          timestamp = t;
15      }
16
17      transaction getLongitude(WeatherRecord@Unowned this) returns int {
18          return longitude;
19      }
20
21      transaction getLatitude(WeatherRecord@Unowned this) returns int {
22          return latitude;
23      }
24
25      transaction getMoisture(WeatherRecord@Unowned this) returns int {
26          return moisture;
27      }
```

```
28
29      transaction getTimestamp(WeatherRecord@Unowned this) returns int {
30          return timestamp;
31      }
32  }
```

Solidity Implementation

bank.sol

```
1   pragma solidity ^0.4.0;
2   import "./money.sol";
3
4   contract Bank {
5       Money mon;
6       address owner;
7       mapping (address => uint) public balances;
8
9
10      constructor() public {
11          mon = new Money(0);
12          owner = msg.sender;
13      }
14
15      function addMoney(uint amount) public {
16          mon.addMoney(new Money(amount));
17      }
18
19
20      function send(address receiver, uint amount) public {
21          require(balances[msg.sender] >= amount, "Not enough money to send.");
22          balances[msg.sender] -= amount;
23          balances[receiver] += amount;
24      }
25
26  }
```

insurancebid.sol

```
1   pragma solidity ^0.4.0;
2   import "./money.sol";
3
4
5   contract InsuranceBid {
6       uint public expirationTime;
7       uint public cost;
8       Money public payout;
9       address public insurer;
10
11      constructor (uint _expirationTime, uint _cost, Money _payout) public {
12          insurer = msg.sender;
13          expirationTime = _expirationTime;
14          cost = _cost;
15          payout = _payout;
16      }
17
18  }
```

insuranceservice.sol

```
1   pragma solidity ^0.4.0;
2   import "./insurer.sol";
3   import "./policyrecord.sol";
4
5
```

```
6  contract InsuranceService {
7      Insurer[] public insurers;
8      address public owner;
9      mapping (address => PolicyRecord[]) public pendingPolicies;
10     mapping (address => PolicyRecord) public activePolicies;
11
12     constructor() public {
13         owner = msg.sender;
14     }
15
16     function requestBids(uint info) public returns (Policy[]){
17         Policy[] memory policies = new Policy[](insurers.length);
18         PolicyRecord[] memory pending = new PolicyRecord[](insurers.length);
19
20         for (uint i = 0; i < insurers.length; i++) {
21             InsuranceBid insBid = insurers[i].requestBid(info); //info for now is just int
22             uint cost = insBid.cost();
23             uint expirationTime = insBid.expirationTime();
24             Money payout = insBid.payout();
25
26             Policy policy = new Policy(cost, expirationTime);
27             PolicyRecord record = new PolicyRecord(policy, payout);
28
29             policies[i] = policy;
30             pending[i] = record;
31         }
32
33         pendingPolicies[msg.sender] = pending;
34
35         return policies;
36     }
37
38     function buyPolicy(Policy policy, Money money, uint i) public returns (Money){
39         require(policy.cost() == money.amount(), "Not enough money to buy policy.");
40         require(policy.currentState() == Policy.States.Offered,
41                 "Policy must be Offered in order to purchase.");
42
43         // assuming i is index of the Policy...
44         PolicyRecord pendingPolicy = pendingPolicies[msg.sender][i];
45
46
47         if (now > policy.expirationTime()) {
48             Money refund = pendingPolicy.refund();
49             // have to find insurer from insurers and give refund...
50             policy.expire();
51             return money;
52         } else {
53             uint cost = policy.cost();
54             Money payment = money.getAmountOfMoney(cost);
55             pendingPolicy.activate(payment);
56             activePolicies[msg.sender] = pendingPolicy;
57             policy.activate;
58             return money;
59         }
60
61     }
62
63     function addInsurer(Insurer insurer) public {
64         insurers.push(insurer);
65     }
66 }
```

insurer.sol

```
1  pragma solidity ^0.4.0;
2  import "./money.sol";
3  import "./insurancebid.sol";
```

```
4  import "./bank.sol";
5
6
7  contract Insurer {
8      Money public mon;
9      Bank public bank;
10     address public owner;
11
12     constructor(Bank _bank, Money _money) public {
13         bank = _bank;
14         mon = _money;
15         owner = msg.sender;
16     }
17
18     // TODO: Calculate bids using some needed information, for now just int
19     function requestBid(uint i) public returns (InsuranceBid) {
20         uint costOfBid = i + 4;
21         uint costOfPayout = i + 6;
22
23         // available for 24 hours
24         uint twentyFourHours = 24 * 3600000;
25         uint expirationTime = now + twentyFourHours;
26
27         // buy token (to ensure money has been given)
28         Money m = mon.getAmountOfMoney(costOfPayout);
29
30         InsuranceBid bid = new InsuranceBid(expirationTime, costOfBid, m);
31
32         return bid;
33     }
34
35
36     function receiveRefund(Money m) public {
37         mon.addMoney(m);
38     }
39 }
```

money.sol

```
1  pragma solidity ^0.4.0;
2  contract Money {
3      uint public amount;
4      address public owner;
5
6      constructor(uint _amount) public {
7          owner = msg.sender;
8          amount = _amount;
9      }
10
11     function addMoney(Money m) public {
12         // validate m's owner somehow?
13         // here we can ensure that the owner of the money isn't
14         // continuously sending money to themselves
15         require(owner != m.owner(), "Can't add the same money to current money.");
16         amount += m.amount(); // have to make amounts public for this
17         delete(m);
18     }
19
20     function getAmountOfMoney(uint _amount) public returns (Money) {
21         require(_amount <= amount, "Requesting too much money.");
22         amount -= _amount;
23         return Money(_amount);
24     }
25
26 }
```

policy.sol

```
pragma solidity ^0.4.0;
contract Policy {
    enum States {Offered, Active, Expired}
    States public currentState;
    uint public cost;
    uint public expirationTime;

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

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

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

policyrecord.sol

```
pragma solidity ^0.4.0;
import "./policy.sol";
import "./money.sol";


contract PolicyRecord {
    Policy public policy;
    enum States {Pending, Active, Expired}
    States public currentState;
    Money money;


    constructor (Policy _policy, Money _money) public {
        policy = _policy;
        money = _money;
        currentState = States.Pending;
    }

    function refund() public returns (Money) {
        require(currentState == States.Pending, "PolicyRecord must be Pending to give refund.");
        currentState = States.Expired;
        Money returnMoney = money;
        delete(money);
        return returnMoney;
    }

    function activate(Money policyCost) public {
        require(currentState == States.Pending, "PolicyRecord must be Pending to activate.");
        money.addMoney(policyCost);
        currentState = States.Active;
    }
}
```

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs*. MIT Press.

[2] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. 2017. Security developer studies with GitHub users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*. 81–95.

[3] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. 2011. The Jabberwocky programming environment for structured social computing. In *User interface software and technology (UIST '11)*. ACM, 53–64. https://doi.org/10.1145/2047196.2047203

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 187–197. https://doi.org/10.1145/581339.581365

[5] 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. https://doi.org/10.1145/1639950.1640073

[6] Paulo Sérgio Almeida. 1997. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*.

[7] American National Standards Institute 1983. *Military Standard Ada Programming Language*. American National Standards Institute. Also MIL-STD-1815A.

[8] American National Standards Institute 1983. *The Pascal Programming Language. ANSI/IEEE 770X3.97-1983*. American National Standards Institute.

[9] 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)*. https://doi.org/10.1007/978-3-030-03596-9_8

[10] 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)*. https://doi.org/10.1007/978-3-662-54455-6_8

[11] 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)*.

[12] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '19)*. 177–210. https://doi.org/10.1145/3344429.3372508

[13] E. D. Berger, S. M. Blackburn, M. Hauswirth, and M. Hicks. 2018. Empirical Evaluation Checklist (beta). http://www.sigplan.org/Resources/EmpiricalEvaluation/

[14] Ilias Bergström and Alan F Blackwell. 2016. The practices of programming. In *Visual Languages and Human-Centric Computing (VL/HCC '16)*. IEEE, IEEE, 190–198. https://doi.org/10.1109/VLHCC.2016.7739684

[15] Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

[16] H. Beyer and K. Holtzblatt. 1997. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Publishers, Inc.

[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)*. https://doi.org/10.1145/2993600.2993611

[18] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Object-oriented programming systems, languages, and applications (OOPSLA '07)*. 301–320. https://doi.org/10.1145/1297027.1297050

[19] 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. https://doi.org/10.1145/1370175.1370213

[20] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. 2011. Checking concurrent typestate with access permissions in plural: A retrospective. *Engineering of Software* (2011). https://doi.org/10.1007/978-3-642-19823-6_4

[21] Alan Blackwell and Margaret Burnett. 2002. Applying Attention Investment to End-User Programming. In *Human Centric Computing Languages and Environments (HCC '02)*. IEEE Computer Society, Washington, DC, USA, 28–30. https://doi.org/10.1109/HCC.2002.1046337

[22] Ann Blandford and Thomas Green. 2008. Methodological Development. In *Research*

Methods for Human-Computer Interaction, Paul Cairns and Anna L. Cox (Eds.). Cambridge University Press, 158–174. https://doi.org/10.1017/CBO9780511814570

[23] Joshua Bloch. 2008. *Effective Java, Second Edition*. Addison-Wesley.

[24] Susanne Bødker and Ole Sejer Iversen. 2002. Staging a Professional Participatory Design Practice: Moving PD Beyond the Initial Fascination of User Involvement. In *Proceedings of the Second Nordic Conference on Human-computer Interaction (NordiCHI '02)*. ACM, 11–18. https://doi.org/10.1145/572020.572023

[25] Michael Bostock and Jeffrey Heer. 2009. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics* 15, 6 (2009), 1121–1128. https://doi.org/10.1109/TVCG.2009.174

[26] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[27] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. *Object-oriented programming, systems, languages, and applications* (2002). https://doi.org/10.1145/583854.582440

[28] John Boyland. 2003. Checking Interference with Fractional Permissions. In *International Conference on Static Analysis (SAS '03)*. https://doi.org/10.1007/3-540-44898-5_4

[29] 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)*. https://doi.org/10.1007/3-540-45337-7_2

[30] Bertrand Braunschweig and Rafiqul Gani. 2002. *Software architectures and tools for computer aided process engineering*. Computer Aided Chemical Engineering, Vol. 11. Elsevier.

[31] Raymond PL Buse, Caitlin Sadowski, and Westley Weimer. 2011. Benefits and barriers of user evaluation in software engineering research. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. 643–656. https://doi.org/10.1145/2076021.2048117

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

[33] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2011. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, 23–32. https://doi.org/10.1145/1985441.1985448

[34] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. 1992. Modula-3 Language Definition. *ACM SIGPLAN Notices* 27, 8 (Aug. 1992), 15–42. https://doi.org/10.1145/142137.142141

[35] Roger D. Chamberlain. 2017. Assessing User Preferences in Programming Language Design. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. Association for Computing Machinery, 18–29. https://doi.org/10.1145/3133850.3133851

[36] Sarah Chasins. 2017. Helena: Web Automation for End Users. http://helena-lang.org/

[37] 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)*. https://doi.org/10.1145/286936.286947

[38] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. *Object-oriented programming, systems, languages, and applications* (1998). https://doi.org/10.1145/286942.286947

[39] 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. https://doi.org/10.1007/978-3-642-36946-9

[40] 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. https://doi.org/10.1145/2688204.2688218

[41] Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Obsidian smart contract programming language. (8 2020). https://doi.org/10.1184/R1/12814202.v1

[42] Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Obsidian vs. Solidity RCT Replication Package. (8 2020). https://doi.org/10.1184/R1/12771074.v1

[43] 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. https://doi.org/10.1145/3276954.3276965

[44] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. 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.

[45] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019. Usability Methods for Designing Programming Languages for Software Engineers. arXiv:1912.04719

[46] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *International Conference on Software Engineering (ICSE '17)*. IEEE Press, 496–506. https://doi.org/10.1109/ICSE.2017.52

[47] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Glacier software and user study replication package. (7 2020). https://doi.org/10.1184/R1/12108693.v1

[48] 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 Transactions on Programming Languages* (2020). To appear.

[49] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring Language Support for Immutability. In *International Conference on Software Engineering (ICSE '16)*. ACM, 736–747. https://doi.org/10.1145/2884781.2884798

[50] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. *Exploring Language Support for Immutability*. Technical Report CMU-ISR-16-106. Carnegie Mellon University.

[51] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019. Smarter Smart Contract Development Tools. *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. https://doi.org/10.1109/WETSEB.2019.00013

[52] Wikipedia Contributors. 2019. Natural Experiment. https://en.wikipedia.org/wiki/Natural_experiment

[53] J. Czerwonka, M. Greiler, and J. Tilford. 2015. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In *International Conference on Software Engineering (ICSE '15)*, Vol. 2. 27–28.

[54] Nils Dahlbäck, Arne Jönsson, and Lars Ahrenberg. 1993. Wizard of Oz studies - why and how. *Knowledge-based systems* 6, 4 (1993), 258–266. https://doi.org/10.1016/0950-7051(93)90017-N

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

[56] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-Aware Session Types for Digital Contracts. arXiv:cs.PL/1902.06056

[57] Sayamindu Dasgupta and Benjamin Mako Hill. 2018. How "Wide Walls" Can Increase Engagement: Evidence From a Natural Experiment in Scratch. In *2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*.

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

[59] 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*.

https://doi.org/10.1007/978-3-662-53357-4_6

[60] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic universe types. In *European Conference on Object-Oriented Programming*. Springer.

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

[62] Yvonne Dittrich, Margaret Burnett, Anders MOrch, and Redmiles David. 2013. End-User Development. In *End-User Development Symposium*. Springer.

[63] J.J. Dolado, M. Harman, M.C. Otero, and L. Hu. 2003. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering* 29, 7 (July 2003), 665–670. https://doi.org/10.1109/TSE.2003.1214329

[64] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

[65] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2002. More Dynamic Object Reclassification: Fickle II. *ACM Trans. on Programming Languages and Systems* 24, 2 (March 2002), 153–191. https://doi.org/10.1145/514952.514955

[66] Joseph S Dumas and Janice Redish. 1999. *A practical guide to usability testing*. Intellect books.

[67] 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. https://doi.org/10.1145/3173574.3174032

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

[69] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *International Conference on Software Engineering (ICSE '14)*. ACM, 632–642. https://doi.org/10.1145/2568225.2568299

[70] K Anders Ericsson and Herbert A Simon. 1984. *Protocol analysis: Verbal reports as data*. the MIT Press.

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

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

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

[74] Ethereum Foundation. 2020. Withdrawal from Contracts. Retrieved February 25, 2020 from https://solidity.readthedocs.io/en/v0.6.3/common-

patterns.html#withdrawal-from-contracts

[75] Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Programming Language Design and Implementation (PLDI '02)*. 12. https://doi.org/10.1145/512529.512532

[76] Ethereum Foundation. 2020. Simple Open Auction. https://solidity.readthedocs.io/en/v0.6.3/solidity-by-example.html#simple-open-auction

[77] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[78] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. on Programming Languages and Systems* 36, 4, Article 12 (Oct 2014), 44 pages. https://doi.org/10.1145/2629609

[79] Google Inc. 2019. Protocol Buffers. Retrieved February 18, 2020 from https://developers.google.com/protocol-buffers/

[80] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. *Object-oriented programming, systems, languages, and applications* (2012). https://doi.org/10.1145/2398857.2384619

[81] 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

[82] Paul Graham. 2001. Five Questions about Language Design. http://www.paulgraham.com/langdes.html

[83] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. In *Object-oriented programming systems, languages, and applications (OOPSLA '18)*.

[84] Paul Green and Lisa Wei-Haas. 1985. *The Wizard of Oz: a Tool for Rapid Development of User Interfaces*. Technical Report UMTRI-85-27. University of Michigan.

[85] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[86] Raymonde Guindon, Herb Krasner, Bill Curtis, et al. 1987. Breakdowns and processes during the early activities of software design by professionals. In *Empirical studies of programmers: Second Workshop*. 65–82.

[87] Jan Gulliksen, Bengt Göransson, Inger Boivie, Stefan Blomkvist, Jenny Persson, and Åsa Cajander. 2003. Key principles for user-centred systems design. *Behaviour and Information Technology* 22, 6 (2003), 397–409. https://doi.org/10.1080/

0144929031000162432 9

[88] Christian Haack and Erik Poll. 2009. Type-based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming*. https://doi.org/10.1007/978-3-642-03013-0

[89] C. Haack, E. Poll, J. Schäfer, and A. Schubert. 2007. Immutable objects for a java-like language. In *European Symposium on Programming*.

[90] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (oct 2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

[91] 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

[92] Dominik Harz and William Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. arXiv:1809.09805

[93] Maurice Herlihy. 2019. Blockchains from a distributed computing perspective. *Commun. ACM* 62, 2 (2019), 78–85.

[94] C. A. R. Hoare. 2009. Null References: The Billion Dollar Mistake. Retrieved February 18, 2020 from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[95] John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. *Object-oriented programming systems, languages, and applications* (1991). https://doi.org/10.1145/118014.117975

[96] Richard C. Holt and James R. Cordy. 1988. The Turing Programming Language. *Commun. ACM* 31, 12 (Dec. 1988), 1410–1423. https://doi.org/10.1145/53580.53581

[97] David J. Houston and Lilliard E. Richardson. 2007. Risk Compensation or Risk Reduction? Seatbelts, State Laws, and Traffic Fatalities. *Social Science Quarterly* 88, 4 (2007), 913–936. https://doi.org/10.1111/j.1540-6237.2007.00510.x

[98] 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)*.

[99] IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from https://www.ibm.com/blockchain/supply-chain/

[100] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. on Programming Languages and Systems* 23, 3 (May 2001), 396–450.

[101] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-centred Ap-

proach to Functions in Excel. In *International Conference on Functional Programming (ICFP '03)*. ACM, 165–176. https://doi.org/10.1145/944705.944721

[102] Kadena. 2019. PACT. https://pact.kadena.io

[103] 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)*.

[104] Gauri Kambhatla, Michael Coblenz, Reed Oei, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. 2019. A Pilot Study of the Safety and Usability of the Obsidian Blockchain Programming Language. *PLATEAU Workshop* (2019).

[105] Anne Marie Kanstrup. 2013. Designed by end users: Meanings of technology in the case of everyday life with diabetes. In *International Symposium on End User Development*. Springer, 185–200.

[106] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Șerbănuță, Yi Zhang, Daniele Filaretti, Virgil Șerbănuță, 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)*.

[107] Mary Beth Kery, Claire Le Goues, and Brad A Myers. 2016. Examining programmer practices for locally handling exceptions. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 484–487. https://doi.org/10.1145/2901739.2903497

[108] Fredrik Kjolstad, Danny Dig, Gabriel Acevedo, and Marc Snir. 2011. Transformation for class immutability. In *International Conference on Software Engineering (ICSE '11)*. ACM Press, 61. https://doi.org/10.1145/1985793.1985803

[109] Gunter Kniesel and Dirk Theisen. 2001. JAC—Access right based encapsulation for Java. *Journal of Software Practice & Experience - Special issue on aliasing in object-oriented systems* 31, 6 (2001), 555–576. http://dl.acm.org/citation.cfm?id=377334

[110] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3, Article 21 (2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[111] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[112] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006). https://doi.org/10.1109/TSE.2006.116

[113] Herb Krasner, Bill Curtis, and Neil Iscoe. 1987. *Communication Breakdowns and Boundary*

Spanning Activities on Large Programming Projects. 47–64.

[114] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. on Database Systems* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[115] William Landi. 1992. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337. https://doi.org/10.1145/161494.161501

[116] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC '11)*.

[117] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *International Conference on Software Engineering (ICSE '06)*. https://doi.org/10.1145/1134285.1134355

[118] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

[119] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *SIGCHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, 1449–1461. https://doi.org/10.1145/2858036.2858252

[120] Eugene Lukash. [n. d.]. Immutables. ([n. d.]). https://immutables.github.io/

[121] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Computer and Communications Security (CCS '16)*. https://doi.org/10.1145/2976749.2978309

[122] Donna Malayeri and Jonathan Aldrich. 2009. Is Structural Subtyping Useful? An Empirical Study. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 95–111. https://doi.org/10.1007/978-3-642-00590-9_8

[123] Robert C. Martin, Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education.

[124] Gary McGraw and Ed Felton. 1999. *Securing Java*. Wiley.

[125] Erik Meijer and Peter Drayton. 2004. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer.

[126] Microsoft, Inc. [n. d.]. Freezable Objects Overview. ([n. d.]). https://msdn.microsoft.com/en-us/library/vstudio/ms750509(v=vs.100).aspx

[127] 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.

[128] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014.

Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, 309–328. https://doi.org/10.1145/2660193.2660231

[129] Rob Moffat. [n. d.]. Pure4J @ImmutableValue Specification. ([n. d.]). http://pure4j.org/concordion/org/pure4j/test/checker/spec/ImmutableValue.html

[130] Mozilla Developer Network. [n. d.]. Object.freeze(). ([n. d.]). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

[131] 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. https://doi.org/10.1109/MC.2016.200

[132] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47 (2004), 47–52. Issue 9. https://doi.org/10.1145/1015864.1015888

[133] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69. https://doi.org/10.1145/2896587

[134] Allen Newell and Stuart K. Card. 1985. The Prospects for Psychological Science in Human-computer Interaction. *Hum.-Comput. Interact.* 1, 3 (Sept. 1985), 209–242. https://doi.org/10.1207/s15327051hci0103_1

[135] W. R. Nichols. 2019. The End to the Myth of Individual Programmer Productivity. *IEEE Software* 36, 5 (2019), 71–75.

[136] Jakob Nielsen. 1993. *Usability engineering*. Academic Press, Boston.

[137] Jakob Nielsen and Thomas K. Landauer. 1993. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 206–213. https://doi.org/10.1145/169059.169166

[138] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *SIGCHI conference on Human Factors in Computing Systems (CHI 1990)*.

[139] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design*. Springer-Verlag, Berlin, Heidelberg, 114–136.

[140] James Noble and Alex Potanin. 2014. On owners-as-accessors. *IWACO Proceedings* (2014).

[141] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*. http://dl.acm.org/citation.cfm?id=646155.679699

[142] The University of Glasgow. 2001. System.IO.Unsafe. https://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO-Unsafe.html

[143] University of Washington. 2019. Professional Master's Program. Retrieved February 18, 2020 from https://www.cs.washington.edu/academics/pmp

[144] Stephen Oney, Brad A. Myers, and Joel Brandt. 2014. InterState: a language and environment for expressing interface behavior. In *User interface software and technology (UIST '14)*. ACM, 263–272. https://doi.org/10.1145/2642918.2647358

[145] Fatih Kursat Ozenc, Miso Kim, John Zimmerman, Stephen Oney, and Brad Myers. 2010. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 2513–2522. https://doi.org/10.1145/1753326.1753707

[146] 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. https://doi.org/10.1109/HCC.2002.1046372

[147] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software engineering with transactional memory versus locks in practice. *Theory of Computing Systems* 55, 3 (2014), 555–590.

[148] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[149] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295 – 341. https://doi.org/10.1016/0010-0285(87)90007-7

[150] Kara Pernice and Kathryn Whintenton. 2017. How to Deal With Bad Design Suggestions. Retrieved February 18, 2020 from https://www.nngroup.com/articles/bad-design-suggestions/

[151] Dewayne E. Perry, Susan Elliott Sim, and Steve M. Easterbrook. 2004. Case studies for software engineers. In *International Conference on Software Engineering (ICSE '04)*. IEEE, 736–738.

[152] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[153] Benjamin C. Pierce and Yitzhak Mandelbaum. 2009. PL Grand Challenges. Retrieved February 18, 2020 from http://plgrand.blogspot.com

[154] Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

[155] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Toonelli, and Michele Marchesi. 2019. A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics. *IEEE Access* 7 (2019).

[156] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming*. 233–269.

[157] L. Prechelt and W.F. Tichy. 1998. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering* 24, 4 (apr 1998), 302–312. https://doi.org/10.1109/32.677186

[158] Qualtrics. 2020. Qualtrics Software. http://www.qualtrics.com/

[159] Apple, Inc. 2020. The Foundation Framework. (2020). https://developer.apple.com/documentation/foundation

[160] BaseX Team. [n. d.]. BaseX. ([n. d.]). http://basex.org

[161] BaseX Team. [n. d.]. Map: remove and check values. ([n. d.]). https://github.com/BaseXdb/basex/issues/1297

[162] Google. [n. d.]. Guava: Google Core Libraries for Java. ([n. d.]). https://github.com/google/guava

[163] Microsoft Corp. 2008. Framework Design Guidelines. Retrieved February 18, 2020 from https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/struct

[164] Mozilla Research. 2015. The Rust Programming Language. Retrieved February 18, 2020 from https://www.rust-lang.org

[165] Oracle Corp. 2019. Secure Coding Guidelines for the Java SE, version 4.0. Retrieved February 18, 2020 from https://www.oracle.com/technetwork/java/seccodeguide-139067.html

[166] Potix Corportation. [n. d.]. ZK Spreadsheet. ([n. d.]). https://www.zkoss.org/product/zkspreadsheet

[167] University of Washington. [n. d.]. IGJ immutability checker. ([n. d.]). http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#igj-checker. Accessed Feb. 8, 2016.

[168] University of Washington. [n. d.]. The Checker Framework. ([n. d.]). http://types.cs.washington.edu/checker-framework/. Accessed Feb. 8, 2016.

[169] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. 2009. Scratch: Programming for all. *Commun. Acm* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[170] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[171] BF Robertson and DF Radcliffe. 2009. Impact of CAD tools on creative problem solving in engineering design. *Computer-Aided Design* 41, 3 (2009), 136–146. https://doi.org/10.1016/j.cad.2008.06.007

[172] Paulette M. Rothbauer. 2008. *The Sage encyclopedia of qualitative research methods*. SAGE.

[173] Grigore Roșu and Traian Florin Șerbănuță. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[174] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style.. In *Conference on LISP and Functional Programming (LFP '92)*. 11. https://doi.org/10.1145/141471.141563

[175] Chris Sadler and Barbara Ann Kitchenham. 1996. Evaluating Software Engineering Methods and Tool—Part 4: The Influence of Human Factors. *SIGSOFT Softw. Eng. Notes* 21, 5 (Sept. 1996), 11–13. https://doi.org/10.1145/235969.235972

[176] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 659–668. https://doi.org/10.1109/TVCG.2015.2467091

[177] 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

[178] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for Safer Smart Contracts. (2019). arXiv:1904.06534

[179] Robert C. Seacord. 2009. *The Cert C Secure Coding Standard*. Pearson Education, Inc.

[180] Robert C Seacord. 2013. *Secure Coding in C and C++*. Addison-Wesley.

[181] 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)*. https://doi.org/10.1145/3360611

[182] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The billion-dollar fix. In *European Conference on Object-Oriented Programming (ECOOP '13)*.

[183] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[184] Ben Shneiderman and Catherine Plaisant. 2006. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In *AVI workshop on BEyond time and errors: novel evaluation methods for information visualization*. ACM, ACM, 1–7. https://doi.org/10.1145/1168149.1168158

[185] 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/

[186] Mats Skoglund and Tobias Wrigstand. 2001. A mode system for read-only references in Java. In *3rd Workshop on Formal Techniques for Java Programs*.

[187] Stack Overflow. 2019. Developer Survey Results 2019. Retrieved February 18, 2020 from https://insights.stackoverflow.com/survey/2019

[188] 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. https://doi.org/10.1145/2661136.2661156

[189] Andreas Stefik and Stefan Hanenberg. 2017. Methodological irregularities in programming-language research. *Computer* 50, 8 (2017), 60–63. https://doi.org/10.1109/MC.2017.3001257

[190] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. 2014. What is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. 223–231. https://doi.org/10.1145/2597008.2597154

[191] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 19.

[192] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. 2011. An Empirical Comparison of the Accuracy Rates of Novices Using the Quorum, Perl, and Randomo Programming Languages. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11)*. ACM, 3–8. https://doi.org/10.1145/2089155.2089159

[193] Kent D. Stewart, Melisa Shiroda, and Craig A. James. 2006. Drug Guru: a computer software program for drug design using medicinal chemistry rules. *Bioorganic & medicinal chemistry* 14, 20 (2006), 7011–7022.

[194] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[195] Jeffrey Stylos and Steven Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. In *International Conference on Software Engineering (ICSE '07)*. https://doi.org/10.1109/ICSE.2007.92

[196] 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)*. https://doi.org/10.1007/978-3-662-44202-9_7

[197] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2015. Searching the State Space: A Qualitative Study of API Protocol Usability. In *International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 82–93. http://dl.acm.org/citation.cfm?id=2820282.2820295

[198] 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)*. https://doi.org/10.1145/2076021.2048122

[199] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). https://doi.org/10.5210/fm.v2i9.548

[200] The Linux Foundation. 2020. Hyperledger Fabric. Retrieved February 18, 2020 from https://www.hyperledger.org/projects/fabric

[201] Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Principles of Programming Languages (POPL '11)*. https://doi.org/10.1145/1926385.1926436

[202] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *Object-oriented programming systems, languages, and applications (OOPSLA*

'05). https://doi.org/10.1145/1103845.1094828

[203] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, 1–17. https://doi.org/10.1145/3133850.3133863

[204] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *International Conference on Software Engineering (ICSE '16)*. ACM, 760–771. https://doi.org/10.1145/2884781.2884849

[205] Carnegie Mellon University. 2019. Master's Programs. Retrieved February 18, 2020 from https://www.cs.cmu.edu/masters-programs

[206] Christopher Unkel and Monica S. Lam. 2008. Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields. In *Principles of Programming Languages (POPL '08)*. https://doi.org/10.1145/1328438.1328463

[207] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. 2015. Cascade: A universal programmer-assisted type qualifier inference tool. In *International Conference on Software Engineering (ICSE '15)*.

[208] A. Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. 1999. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies* 51, 1 (1999), 31 – 70. https://doi.org/10.1006/ijhc.1999.0268

[209] John Venable, Jan Pries-Heje, and Richard Baskerville. 2012. A Comprehensive Framework for Evaluation in Design Science Research. In *Design Science Research in Information Systems. Advances in Theory and Practice*. 423–438. https://doi.org/10.1007/978-3-642-29863-9_31

[210] Bill Venners and Bruce Eckel. 2004. A Conversation with Anders Hejlsberg, Part VIII. (Feb. 2004). http://www.artima.com/intv/choicesP.html

[211] June M. Verner, Jennifer Sampson, Vladimir Tosic, N.A. Abu Bakar, and Barbara A. Kitchenham. 2009. Guidelines for industrially-based multiple case studies in software engineering. In *International Conference on Research Challenges in Information Science*. IEEE, 313–324. https://doi.org/10.1109/RCIS.2009.5089295

[212] Willemien Visser. 1987. Strategies in programming programmable controllers: A field study on a professional programmer. In *Empirical Studies of Programmers: Second workshop (ESP2)*. 217–230. https://hal.inria.fr/hal-00641376/document

[213] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP 20: ERC-20 Token Standard. Retrieved February 18, 2020 from https://eips.ethereum.org/EIPS/eip-20

[214] Philip Wadler. 1990. Linear types can change the world. In *Programming concepts and methods*, Vol. 2. 347–359.

[215] Diane B. Walz, Joyce J. Elam, Herb Krasner, and Bill Curtis. 1987. A Methodology for Studying Software Design Teams: An Investigation of Conflict Behaviors in the

Requirements Definition Phase. In *Empirical Studies of Programmers: Second Workshop*. 83–99.

[216] Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2017. Design and Implementation of Concurrent C0. arXiv:cs.PL/1701.04929

[217] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Symposium on New Ideas in Programming and Reflections on Software (Onward! 2017)*. 1–17. https://doi.org/10.1145/3133850.3133863

[218] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *Foundations of Software Engineering (FSE '15)*. https://doi.org/10.1145/2786805.2786816

[219] 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)*.

[220] Serdar Yegulalp. 2018. Rust language is too hard to learn and use, says user survey. https://www.infoworld.com/article/3324488/rust-language-is-too-hard-to-learn-and-use-says-user-survey.html

[221] Robert K Yin. 2017. *Case study research and applications: Design and methods*. Sage publications.

[222] Mariam Yusuf. 2018. A Comprehensive List of Blockchain Platforms. Retrieved February 18, 2020 from https://www.technoduet.com/a-comprehensive-list-of-blockchain-platforms/

[223] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kielun, and Michael D. Ernst. 2007. Object and reference immutability using Java generics. In *Foundations of Software Engineering (FSE '07)*. ACM, 75–84.

[224] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic Java. (2010). https://doi.org/10.1145/1932682.1869509