User-Centered Design of Principled Programming Languages

Michael Coblenz

Thursday 6th September, 2018

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 Murphy (University of British Columbia)

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

Keywords: programming language design, human-centered design, blockchain programming, smart contracts, immutability, Obsidian, Glacier

Abstract

Programming languages are designed to facilitate software development by people. 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 are existing languages.

The hypothesis is supported by two language design projects, which integrate a variety of human-centered design approaches into 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 of 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.

Obsidian is 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. Correctness of blockchain software is particularly critical because blockchains are being proposed for high-stakes applications, such as financial transactions, and because smart contracts are immutable and therefore cannot be fixed once deployed.

From two 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. By conducting formative studies with programmers, I am refining the language design to make Obsidian more effective for programmers.

I will finish a formal specification of the core of Obsidian and prove appropriate soundness theorems. I propose to complete the implementation of Obsidian, resulting in a practical language for writing programs on the Hyperledger Fabric blockchain platform. I propose to continue the studies of Obsidian with a quantitative study comparing my proposed approach to typestate to a traditional approach, which is used in existing research languages. I propose to evaluate Obsidian by doing case studies to show that Obsidian is appropriate for implementing representative blockchain applications, such as parametric insurance. Furthermore, I plan to run a user study comparing Obsidian to Solidity, the domain-specific language in most-common use for implementing blockchain applications. I hope to show that, after a learning period, programmers who use Obsidian are more effective at writing correct blockchain applications than programmers who use Solidity. The goal is not just to design Obsidian itself but to show by example how language designers can take users directly into account at early stages of language design.

Contents

Chapter 1

Introduction

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, there has been less focus in the literature on how to analyze the impact of specific language designs on programmers. When software engineers and programmers choose which language to use for a project, and when language designers make decisions, there is little 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* [45]; Stefik et al. 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.

This thesis takes the view that although summative comparisons of different languages can be helpful, finer-grained, lower-cost methods are needed in order to be practically applied to real language design processes. Language designers are faced with making a large number of design decisions, and although all of them may impact the user, it is impractical to design and implement a complete language and then do a user study in order to evaluate each design decision – especially since many language design decisions depend on other decisions. Instead, it is prudent to use *formative* design methods in order to gather data and build theories about how particular design decisions affect users. Indeed, it is the development of *theories* that allows designers to generalize from evidence about particular design decisions in prior contexts to make decisions about new design questions in new contexts without having to run high-cost controlled trials. Further research will consider these theories in new contexts and refine them to build an empirically-validated understanding of how language design decisions relate to users.

In my dissertation research, I am developing methods [15] for designing programming languages that take users into account directly at every phase of the language design process. These methods are adapted from the human-computer interaction literature to make them appropriate in a programming language design context. For example, 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. Finally, 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.

In the service of developing new language design methods, I am designing and implementing *Obsidian*, a new programming language intended for developing blockchain programs (sometimes called *smart contracts* [51]). Obsidian serves as a testbed for new language design methods as well as a test case for the human-centered language design approach.

Blockchain programming is a particularly compelling context in which to study language design. Although

several languages are in common use, including Solidity and Go, bugs in Solidity programs have been exploited by hackers to steal money. 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; violating this trust with incorrect behavior defeats the point of the platform.

In the design of Obsidian, I take the view that the language should provide safety guarantees that are relevant and useful for programs that are appropriate on the blockchain platforms. However, full correctness relative to a specification will require a separate verification effort, which might in the future be layered on top of Obsidian. Obsidian focuses on providing strong guarantees in a way that is accessible to programmers without special training, such as a background in software verification research, because currently software verification is so expensive that it is only practical for very carefully-selected applications. On the other hand, type systems 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.

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 terminate the program). Obsidian only moves checks from compile time to runtime when this provides substantial usability advantages.

Designers and engineers in a variety of domains have a long history of using cost-effective methods to make decisions. For example, *heuristic analysis* [39] has been widely adopted by interface designers. Nielsen's heuristics do not provide a guaranteed path to a good user interface; rather, they constitute a method that designers can use to obtain insight about their proposed designs. Like heuristic analysis, I propose a suite of user-oriented methods that enable language designers to navigate the design space in a way that is more likely to result in a high-quality design. Of course, use of good methods does not guarantee the quality of the created product. In the end, the language must be evaluated using traditional, summative means, such as formal proofs of soundness, randomized controlled trials with appropriate participants, case studies, and deployment in a user community.

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. 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 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.

Chapter 2

Thesis Statement

The thesis is that combining theoretical and human-centered methods in the design and evaluation of programming languages leads 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 two language design projects, which integrate a variety of human-centered design approaches into the programming language design process.

2.1 Glacier

Glacier [14] 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 [12, 13]. 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 can be leveraged effectively by programmers in a lab.

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.

2.2 Obsidian

Obsidian is 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 blockchain programs cannot be directly modified after deployment. This means that bugs can be impossible to fix.
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 verification mechanism, which is so far too difficult and expensive to be practical for most programers, Obsidian rests on 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* [1] and *linear types* [54] 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 spaces for managing this combination is particularly complicated; Obsidian shows how to combine these ideas in a coherent, understandable way.

The rest of the thesis proposal describes work to date on Obsidian and the remaining work that is proposed. In summary, I propose to complete the Obsidian language design and implementation; prove formally that the Obsidian core satisfies appropriate soundness properties; conduct case studies showing that Obsidian is applicable to the kinds of programs that people want to write for blockchain applications; and conduct a user study comparing Obsidian to an existing language used for writing blockchain applications, such as Solidity [24]. Work on Glacier was published at ICSE 2016 [12] and ICSE 2017 [14]; I do not plan any additional work on Glacier for this thesis.

2.3 Technical contributions

Glacier represents a new point in the design space of immutability systems. Glacier's design shows it is possible to obtain both useful expressiveness and simplicity in one design. Although Glacier does not permit expressing every kind of immutability that has ever been proposed, it provides strong, useful immutability properties that users can actually use effectively.

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 will provide 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; the solution will represent a novel approach to compromise among expressiveness and simplicity.

2.4 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. 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, we can study design decisions with fewer confounds. Like other methods, of course, this method has limitations. The design decision 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 (§4.3.2).

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 this problem with several techniques. First, showing superiority 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." What is important is that the methodology allows a researcher to characterize the effects of a language design choice on a particular population. 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.

Third, I triangulate among many different methods. For example, one can do interviews or surveys to assess to what extent programmers believe an approach might be helpful or to what extent they encounter a particular problem. Then, one can do a qualitative study to obtain in-depth details about the usability of a particular design, and finally a quantitative study can try to show statistically significant benefits. If 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 de to variance or some other problem, qualitative evidence of benefit suffices in certain cases to motivate future development or deployment.

Chapter 3

Related Work

This section represents a partial summary of some related work pertaining to human-centered programming language design; the thesis itself will contain considerably more detail and will be more complete. The chapter on Obsidian (§4) includes a separate section (§4.2) describing work pertaining more directly to the proposed research project.

In general, related areas of work include work on empirical studies of programmers and programming environments, such as that by Andy Ko, Brad Myers, and John Pane [37]. Some languages focus on novices, such as Scratch [33]; others focus on children, to avoid bias according to prior language experience [40]; others focus on debugging tasks [30].

A line of work by Stefen Hanenberg and Andreas Stefik has explored empirical evaluation of languages and their features. For example, Kurtev et al. conducted a qualitative study on Quorum [31] to identify usability problems faced by novices. Wilson et al. used Mechanical Turk to elicit feedback about behavior in esoteric situations in a theoretical programming language, finding low consistency and low consensus [56]. Our work focuses on exploring design candidates that theory suggests may be good designs. Quantitative studies have considered specific design questions, such as static vs. dynamic types [29].

Chapter 4

Obsidian

4.1 Background

4.1.1 Blockchains

Blockchain applications run on blockchain platforms, such as Ethereum [22] and Hyperledger [52]. These platforms provide a *single-machine* abstraction on which blockchain programs run. From the perspective of a program running in the blockchain, there is only one machine and one global state. This is in contrast with many other kinds of distributed computer systems, in which programs must directly be aware of other nodes. But, in contrast with most other distributed systems, blockchains do not aim to provide scalability or performance advantages over single-host systems; instead, all of the additional nodes on the network serve to protect against tampering, dishonest nodes, and failures.

The blockchain platform runs a *consensus algorithm*, which ensures that the nodes eventually agree on the global state. However, the consensus algorithm depends on the platform, and some platforms support more than one possible consensus algorithm. The choice of consensus algorithm represents a tradeoff between robustness to attack and efficient consensus. Typically, private blockchain networks, in which a small number of stakeholders control all of the nodes and potentially also control access, optimize for performance, whereas public blockchain networks, which make fewer assumptions of trust of the participants, optimize more for robustness.

Blockchain systems host programs, sometimes called *smart contracts* [51]. These programs can maintain *state*. In this document, I avoid the use of the term *smart contract* because I have found that it tends to evoke assumptions about relationships between programs and legal agreements, which are outside the scope of a blockchain programming language. These programs are deployed on the blockchain; once deployed, they expose *transactions*, which clients can invoke to query or mutate the state stored on the blockchain. This architecture is analogous to that of a distributed database, in which a system maintains persistent state that can be queried and mutated via defined interfaces by external clients.

Blockchain programs can be thought of as general-purpose programs, but there are some additional constraints relative to traditional programs. First, blockchain programs must be entirely *deterministic* given the transaction inputs. Otherwise, different nodes may compute different state updates, which would cause consensus to fail. Second, because of the massively redundant system design, executing code is substantially more expensive than in a traditional system; each computation must be performed independently on each node of the system and all state must be replicated on every node. This leads to patterns in which rather than storing large records on the blockchain, the blockchain stores only cryptographic hashes of the large records and references to off-chain storage locations. Then, the off-chain storage locations need not be trusted to not

tamper with the data, since the any tampering would cause the data to become inconsistent with the on-chain hash.

In public blockchains, such as Ethereum, hosts must be provided an incentive to process transactions and store state. In these systems, transaction fees pay for these costs; the fees typically depend on both the amount of storage required and the amount of computation required. This naturally incentivizes authors of applications to write efficient code. This also results in a problem of resource estimation, since the invoker of a transaction typically wants to know how expensive the execution will be.

4.1.2 Application domain and core features of Obsidian

Blockchains provide an abstraction of a single-threaded machine, hiding the details of the distributed system from programs running on the blockchain. This enables Obsidian to be primarily designed around the *applications* that have been proposed for blockchain platforms. I began the Obsidian project by studying applications that various authors have proposed for blockchains, including informal interviews in some cases. Since then, Elsden et al. published a taxonomy of blockchain applications [21]. I made two observations about many proposed blockchain applications, which form the basis of the core features of Obsidian.

First, many applications are highly stateful, with different states supporting different operations. There is a history of bugs in software in general caused by incorrect manipulation of state, such as attempting to read from a closed file. The DAO exploit [28] can be thought of as a case of unsafe state manipulation. In it, a reentrant call allowed an attacker to drain funds from the contract. One approach to preventing this problem is to require that external invocations are only permitted while the contract is in a consistent state so any reentrant invocations will have their preconditions met. I manually analyzed a sample of the largest Solidity contracts on GitHub and observed that some of the more complex contracts are implemented as state machines. For example, one can obtain a list of Solidity files that are at least 10 kB long[1]. As of 11/2017, there were 125 such files. Examples include escrow with states `{Created, Paid, Delivered, Finished, Disabled}` and voting: `{Uninitialized, Joining, Secret, Open, Tallied, Terminated}`. This is not the first observation that smart contracts are often stateful; the Solidity documentation says "Contracts often act as a state machine" [23].

There is separate evidence that programmers working with stateful interfaces spend a significant portion of their time understanding the relevant state machines [50]. We hypothesize that this leads to a significant number of bugs, when the programmers make mistakes understanding complex, stateful interfaces and using them correctly. Addressing this while considering our goal to detect as many bugs at compile time as possible, Obsidian is a *typestate-oriented* [1] language. As such, it lifts dynamic state into static types, enabling the compiler to reason about potential object protocol violations. For example, in a hypothetical Bond implementation, the compiler verifies that `payInterest` is only invoked on bonds in `Sold` state.

There are many different ways of designing typestate systems. Some typestate systems, such as Plaid [48], allow complex hierarchies of states to be characterized in typestate. However, our survey of blockchain programs suggested that these contracts tend to only require one named state at a time. Making this assumption significantly simplifies the language—particularly its type system—and offers an appropriate starting point for our first user study of typestate systems.

Second, many applications track important resources, such as virtual currency. A common pattern for mitigating the risk of accidental resource loss in Solidity programs is to include an "escape hatch": when something unexpected happens, the contract reverts to a safe state in which a trusted party can withdraw the remaining funds [34]. Otherwise, the funds might stuck in the contract forever. The assumption is that the

[1]Query: `extension:sol "pragma solidity" size:>10000 NOT SafeMath`

trusted party will deploy a new contract that includes a bug fix and deposit the funds into the new contract. However, this reliance on a trusted party is antithetical to the goals of blockchain infrastructure, which attempts to preclude the need for a trusted administrator.

Linear types [54] have been proposed to provide static guarantees regarding objects that are consumable, unlike traditional objects, which can be re-used arbitrarily. Linear types are typically used to facilitate static reasoning about resources, such as money, and other kinds of static properties that can change as a result of operations. Obsidian uses linear types to represent resources in order to provide static guarantees about resource manipulation. Loss of resources in contracts has been a common concern for programmers of blockchain applications. In Obsidian, resources are unlike other kinds of objects because instantiation and destruction are restricted. For example, virtual currency might be represented with a resource object, preventing a program implementing a virtual wallet from accidentally losing virtual currency. Resources can also be used in an application-dependent way; for example, a "token" object could represent the right to vote in a particular blockchain-based election. A voter could receive ownership of a token after gaining an appropriate stake and vote by submitting their token (transferring ownership to the organization or election contract). By doing so, an application can ensure that each voter can only vote a designated number of times.

4.1.3 Permissions

Typestate-oriented programming and resources are complementary features. Linearity is a type-theoretic framework for reasoning about *both* resources and typestate because unlike with traditional types, typestate can change when operations occur. Consider the problem of aliasing, which occurs when more than one reference exists to a given object. Since Obsidian is designed for long-term management of mutable state, objects in Obsidian can be mutable, and in particular can have mutable state. If a reference to a mutable-state object includes a static state guarantee, then there must be no other references to the object through which the state can be modified (otherwise a mutation through an alias could violate the state guarantee). This results in references with *linear* semantics (in which operations can change the static guarantees) independent of the separate requirement for linearity for resources. For example, consider this code, which reflects an early version of Obsidian:

```
1 transaction takeBond (Bond@AvailableForSale bond)
2    {...}
3 void test () {
4    Bond@AvailableForSale bond = ...;
5    takeBond (bond);
6    takeBond (bond); // ERROR
7 }
```

On lines 1 and 4, `Bond.AvailableForSale` provides a static type guarantee that the referenced object is a `Bond` in `AvailableForSale` state. The error is because the typestate-guaranteeing reference `bond` is consumed on line 5; because the linear guarantee has been given to `takeBond`, the typestate guarantee is no longer available in line 6.

It is not obvious that line 6 should give an error; perhaps instead, `takeBond` should "give back" the reference when it returns, or perhaps the existence of the guarantee should force the object to be dynamically state-immutable, or perhaps static typestate specifications should only be available on objects that are statically state-immutable. Even these approaches are likely too restrictive because they do not allow for non-typestate-specifying and typestate-specifying aliases to an object to exist at the same time. Instead, Obsidian includes a *permission system* [10] that allows users to express many different kinds of constraints.

Name	Example	Meaning
Shared	Bond@Shared	All aliases permit mutation but none specify typestate
ReadOnlyState	Bond@ReadOnlyState	Reference cannot be used to mutate state; reference includes no typestate specification. Can co-exist with typestate-bearing references.
Typestate-bearing	Bond@AvailableForSale	Owning reference, i.e. there can be only one reference to a given object that specifies typestate. Obsidian also supports typestate unions, which mean that the referenced object may be in any of the given states.
Owned (no typestate)	Bond@Owned	Exclusivity without typestate specification. Useful when typestate is statically unknown or irrelevant (e.g. `Money` may not have any named states).

Table 4.1: Permissions in Obsidian

The question of how to combine resources and typestate in one language in a *usable* way is critical to the design of Obsidian. Either feature alone might be too complex for users to use effectively; the two together might be overwhelming. Sunshine found in a study of Plural that participants were confused by the combination of access permissions and typestate annotations [47]. In order to address the question of the relationship between linearity in the context of resources and linearity in the context of typestate, I developed a taxonomy of permission systems to explore the design space. Because of the large size of the space, we focused our exploration on the portions that are reasonable candidates for the design of Obsidian. Using this taxonomy, we decomposed the essential capabilities relevant to resources and typestate to form a set of permissions that unify both sets of features.

Obsidian encodes exclusivity in *ownership*. The type system guarantees that there is at most one owning reference to an object at a time. If the owning reference is to a *resource*, then the owning reference cannot go out of scope; instead, ownership must be assigned to a field, passed to a new owner, or ownership must be explicitly released via a `disown` operation. Only owning references can include typestate specifications, since these references can be used to mutate state; other aliases to objects that have typestate-bearing references cannot be used to modify state. The permissions available in Obsidian are shown in Table 4.1.

4.2 Related work

Authors have investigated common causes of bugs in smart contracts [32, 20, 3] and worked on applying formal verification [7]. Our work focuses on preventing bugs in the first place by designing languages in which programs never have specific bugs that would otherwise be commonplace. Other blockchain programming languages include Flint [44], Vyper [25], and Ergo [42]. Flint represents assets with linear types but does not include a notion of typestate, instead focusing on mutation control via capabilities. Vyper is intended to be simpler than Solidity but does not have linear types. Ergo focuses on a formal relationship between the natural language legal text of a contract and computational properties of the contract's implementation in a program. Some blockchain platforms expect their users to program in general-purpose languages. For example, Corda [43] is designed around the JVM and expects users to program in Kotlin.

The theory of linear types, which facilitate reasoning about *state*, have been studied in depth since Wadler's 1990 paper [54]. Ancona et al. wrote a book describing the history of behavioral types [2], which are types

that describe behavior of systems. In brief, there has been a long line of development around *session types*, which facilitate reasoning about types of channels and processes in concurrent settings, starting with Caires and Pfenning [11]. This initial work has been extended to support shared channels (Balzer and Pfennning) [5] and resource analysis (Das et al.) [17, 18]. Balzer and Pfenning described relationships with object-oriented programming [4]. Session types have been implemented in a research language, Concurrent C0 [55]. Although session types are a promising approach for specifying linearity, in this work I focus on the typestate family of approaches because I hypothesize that typestate reflects a more usable formulation of linearity for users in an object-oriented context. In the specific context of blockchain programs, although the interaction between blockchain software and client (off-chain) software is distributed, the programs running inside the blockchain run in a serial context rather than a parallel one. Obsidian is primarily focused on obtaining safety guarantees within the blockchain.

Typestate originated in a 1986 paper by Strom [46]. Although this paper was written without benefit of a modern formalism to express linearity, many of the general goals are the same. DeLine investigated using typestate in the context of object-oriented systems [19], finding that subclassing causes complicated issues of partial state changes; we avoid that problem by not supporting subclassing. Plural [8] and Plaid [48] are the most closely-related systems in terms of their design goals, but neither language integrated user studies into the design process and neither was intended for a blockchain context. Sunshine et al. showed typestate to be helpful in documentation when users need to understand object protocols [49], but that study (and others on typestate) did not ask participants to write code that used typestate. The designers of Plural conducted summative case studies themselves but did not invite participants for user studies [9]. The notion of typestate was captured in a formalism by Garcia et al. [26]; the Obsidian formalism is based on that approach.

Despite the above work, linear types have not been adopted in many practical programming languages. Rust [36] is one exception, using a form of linearity to restrict aliases to mutable objects. This limited use of linearity did not necessitate the language to support as rich a permission system as Obsidian will. Alms [53] is an ML-like language that supports linear types. As with Plural, the language designers did the case study themselves rather than asking others to use their system.

4.3 Work to date

4.3.1 Initial design work

Initial design work focused on combining the concept of typestate, which has been explored in a variety of contexts [46, 1, 19, 38], with the concept of linear types, which has also been well-explored [54]. However, no typestate-oriented languages have been widely deployed, and those that have been developed were used only by their authors. In general, programming languages researchers tend to focus on expressiveness and abstraction elegance, and as a result, little was known about how effectively programmers who were not researchers could use these systems. Rust [36] is one popularly-used language that incorporates ideas of linearity, but the focus is on mutation and alias control, not on correct manipulation of resources, and Rust does not integrate ideas of typestate explicitly.

4.3.2 Iterative design process

The Obsidian design process includes formative human-centered methods in order to improve the chances that the final language will be as effective as possible for programmers. With my collaborators, I conducted formative studies of prototypes of Obsidian.

Tasks	Research questions
Invent a language to specify the system, using the *natural programming* elicitation technique [37]	How do people naturally solve this kind of state-oriented problem?
Update their implementation according to a given state transition diagram	How do people already represent state machines and transitions?
Add state transitions to a partial Obsidian program	What syntax do people naturally use for state transitions?
Practice several state transition syntaxes	Can people use the alternatives that we have designed effectively?
Select one state transition syntax and use it for the voter registration program	Which approach do people prefer, and what happens when they use it?

Table 4.2: Tasks in Exercise 1 of Barnaby et al. [6]

Celeste Barnaby and I previously conducted formative studies on the usability of several aspects of Obsidian [6]. Key research questions we investigated included:

- Are states and path-dependent types a natural way of approaching the challenges that arise in blockchain programming?
- Which (if any) of our proposed ways of presenting states, state transitions, and path-dependent types is most understandable and usable by programmers?
- Are people able to effectively use and understand path-dependent types as they are implemented in Obsidian?

In that study, we gave participants two different exercises. The first, which focused on states, gave participants a description of a voter registration system. Tasks are shown in Table 4.2.

From seven participants who were given the state transition exercise, our observations included:

- Without any language support, users typically store state explicitly, but may do so in a way that permits inconsistent state. They sometimes failed to implement the code correctly because they did not check state at the right times.
- When asked to write state transitions, participants were not sure what to assume about state transitions, e.g. which variables are in scope at different times or which transactions were available at any given point.
- Among the options for state transitions, state constructions seemed to be preferred, but this preference may not have a relationship with actual performance on tasks.

The second exercise explored *path-dependent types*, which are used to distinguish between types that are dependent on particular objects. For example, we might like to distinguish between two different kinds of `LotteryTicket` types, each corresponding to a `Lottery` object that issued them. Briefly, we found that users were surprised at our use of nested contracts for this purpose; in the future, if we include path-dependent types in Obsidian, we will use a more explicit way to specify them.

With Jenna Wise, I also studied:

- How should users specify cleanup of old fields when a contract transitions to a new state, which may lack some of the old fields?
- What permissions system should Obsidian support to restrict aliasing in order to provide typestate

guarantees? How do users reason when they need to think about typestate and permissions?

We gave users two sets of exercises. In the first set of exercises, we provided a state transition diagram for a `Wallet` object, which had four states, each of which held some combination of money and a license. Participants were given several approaches to specify state transitions. We were interested in which approaches seemed most natural and which seemed most error-prone. Generally, we found that participants expected to prepare resources for the state transition *before* transitioning states, but all of the participants were able to use all of the approaches effectively. However, participants were consistently confused by an approach that returned leftover resources in a collection after making a transition (for example, from transitioning from `HasMoney` to `Empty` requires disposing of the leftover money).

We also found that participants expect to release resources before transitioning rather returning a collection. These results together informed the design of the state transition syntax for Obsidian; objects are allowed to temporarily have fields whose typestates are inconsistent with their declarations as long as the typestates match the declarations at the ends of methods.

In the second set of exercises, we explored the space of permissions designs, making a collection of interesting observations that we used to modify the design of the language. A key technique here was to back-port the language design decisions into the context of Java so that we could study the decisions in isolation, without requiring the participants to learn a whole new language (and thereby confounding the results with the other language design decisions). We cast a prototype permissions system into Java as Java annotations. Then, we gave participants a collection of tasks pertaining to a very small (163-line) medical records system.

We asked the first two participants to identify a bug in the program in which a prescription could be refilled more times than was specified. This was a hard problem, even given the length of the program. We gave the participants 30 minutes to find the bug; one participant found the bug at the end of the 30 minutes and the other did not find the bug at all. We took this as sufficient evidence that identifying the bug is difficult and worth preventing with a type system.

We asked participants to fix the bug using ownership; surprisingly, they found this very difficult, although the fix required only a small code change. Among our observations is the fact that several participants had great difficulty thinking about ownership as a *static* concept. For example, one participant wrote `if (@Owned prescription)` in an attempt to check ownership dynamically. Other participants assumed the compiler could reason about the code as well as they could, for example expecting sophisticated interprocedural analyses rather than a typechecker.

Some participants were confused about when ownership is transferred. Though we had intended to make this clear with type specifications on interfaces, this did not suffice. I am modifying the language to support static typestate and ownership assertions, e.g. `[prescription@Owned]` to indicate that at that point the reference has ownership.

Although the above is only a partial explanation of the studies that have been done so far, it is intended to give the reader some examples of how I have used formative studies to ground the design of Obsidian in data from users even without a complete Obsidian implementation (and without requiring the users to learn the entire language).

Topic	Core language	Surface language
Restrictions	A-normal form	Arbitrary expression nesting
Sequencing	let-bindings	Statements
Field definitions	Repeated identically in each state in which they are available	Defined in one place with an accompanying list of applicable states
Field initialization	Contract and state fields syntactically separated	Contract and state fields syntactically combined
Types	All types represent contracts or `Void`.	Primitive types, including int and string, are supported.

Table 4.3: Key differences between the Obsidian core language and the Obsidian surface language

4.4 Proposed work

4.4.1 Formal work

A formal specification draft for the core of Obsidian is already in progress (see Appendix A), so far including only a static semantics. I will extend the specification to include a dynamic semantics as well. In addition, the thesis will include a paper-based proof of type soundness. Important soundness properties relevant to Obsidian will also be stated and proved. *Exclusive Ownership* defines what ownership means and gives a soundness criterion; *typestate soundness* defines what typestate specifications mean regarding program heaps. These propositions are *draft* propositions at the moment, since formalization of Obsidian is not yet complete.

Proposition 1 (Exclusive Ownership)**.** *In any heap that corresponds to a state of a well-typed Obsidian program, if* $\Delta \vdash x : C.Owned \dashv \Delta'$ *or* $\Delta \vdash x : C.\overline{S} \dashv \Delta'$ *then if y is a different variable than x and y references the same object as x, then neither* $\Delta \vdash y : C.Owned \dashv \Delta'$ *nor* $\Delta \vdash y : C.\overline{S'} \dashv \Delta'$ *is derivable for any* $\overline{S'}$*.*

Proposition 2 (Typestate soundness)**.** *In any heap that corresponds to a state of a well-typed Obsidian program, if* $\Delta \vdash x : C.\overline{S} \dashv \Delta'$*, then* $\exists s \in \overline{S}$ *such that the object referenced by x is in state s of contract C.*

4.4.2 Formative studies

I plan to combine the results of the previous studies on ownership, permissions, and state transitions with the results of ongoing work regarding permissions and submit the results to a journal or conference. In the first phase of that work, which is in progress, I am gathering qualitative data on several different possible permissions systems. In the second phase, I will deploy an experiment based on the first phase to quantitatively compare the refined design to an initial design, with the hope of showing that the improved design is easier for people to use effectively when given small programming tasks.

4.4.3 Surface language

The Obsidian surface language is based on the Obsidian core language but is designed to be substantially more convenient for programmers. Key differences between the surface language and the core language are shown in Table 4.3.

4.4.4 Implementation

In order to practically provide a realistic case study of Obsidian, it was necessary to choose a specific blockchain platform for Obsidian to support initially. Ethereum and Hyperledger Fabric currently seem to be the most mature blockchain platforms in use. For practical reasons, Obsidian supports Hyperledger Fabric initially. Deploying programs to Fabric involves translating Obsidian to either Java or Go, which are compiled with standard tools and can execute in Docker containers. In contrast, Ethereum defines its own virtual machine (the EVM); targeting it involves either compiling to EVM bytecode or compiling to Solidity or Yul. Yul is an immature, novel language; Solidity is object-oriented but has some surprising properties that make the translation from Obsidian to Solidity nontrivial. In contrast, the translation from Obsidian to Java is relatively straightforward. Furthermore, I have contacts at IBM, where Fabric is being developed, making it more likely that I can get help when needed. Hopefully the relevance to IBM will make adoption and further case studies more feasible.

The Obsidian compiler translates Obsidian code to Java and generates a protobuf specification for the archive format of all of the contracts and their interfaces. Then, it uses javac to compile the generated Java code, which interacts with Fabric APIs. This prevents the programmer from needing to specify archive or wire formats for any of their data structures; they are synthesized automatically from the Obsidian code.

Architecture and interfaces with off-blockchain code

Figure 4.1 shows the architecture of the compiler. The compiler takes Obsidian programs as input, generates Java code, and then invokes the Java compiler and `jar` to generate a jar file.

Blockchains provide services, similar to web services, that can be invoked externally (i.e. from code that does not run inside the blockchain). Each contract exposes an interface that specifies what operations it supports. For example, Hyperledger Fabric peer nodes support invocations via a REST API [16]. Although this approach is flexible and allows arbitrary external invocations of the deployed blockchain contract, it is unsafe in that the contract's interpretation of the input may differ from the intended meaning; this is a side effect of round-tripping arbitrary data structures through text and the fact that two different programming languages may be used to represent the data. Ethereum goes as far as limiting the API to including only primitive objects, such as integers and arrays; any objects must be unpacked and sent as collections of primitives.

In contrast to the existing approach, Obsidian reduces the potential for bugs in the client/blockchain interaction by allowing clients to be written in the same language as the server. The compiler can be executed in *server mode* and in *client mode*; generated client programs take as input an IP address to connect to, and run a `main` transaction, which takes as input a remote object. Then, the client can operate on the remote object via a mechanism analogous to RMI.

In this proposal, it is assumed that clients of Obsidian blockchain programs will also be written in Obsidian. However, this is not a permanent limitation of the Obsidian design. In the future, it would be possible to add *Yul* as a target for the Obsidian compiler. Yul is the intermediate language used by the Solidity compiler and compiles down to both eWASM (a restricted subset of WebAssembly) and EVM (Ethereum Virtual Machine). By doing so, Obsidian could support both browser-based web clients for Obsidian blockchain programs as well as deploying Obsidian programs on the Ethereum platform.

Tooling

Although language tooling is not a focus of this thesis, for practical reasons of usability it will be necessary to expend some effort on tooling to make writing Obsidian code convenient, including a text editor plugin to

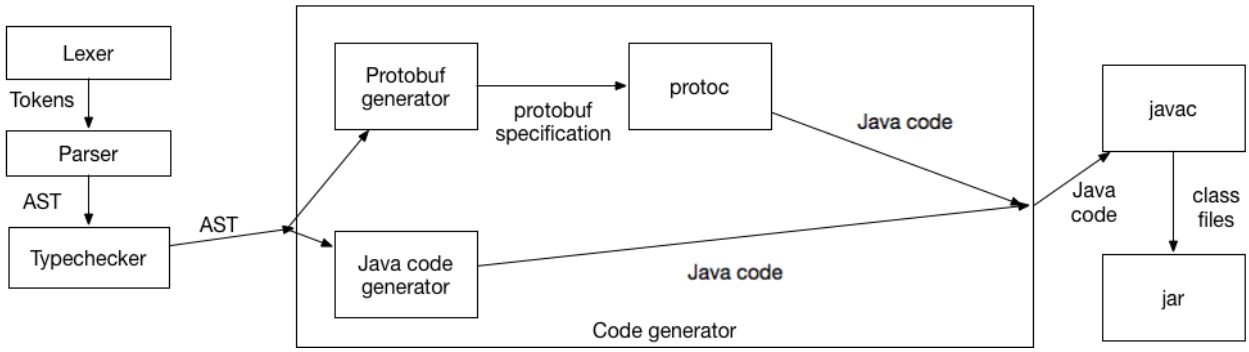

Figure 4.1: Runtime architecture of the Obsidian compiler

provide the customary syntax highlighting features. In future research, however, I would like to explore how tooling can address potential difficulties users may encounter when using Obsidian's novel type system.

4.4.5 Case studies

Case studies address two major objectives. First, case studies are an opportunity to gather qualitative data about the experience of using Obsidian on a real problem. What aspects of Obsidian promote effective software development in domains of interest, and where does Obsidian fall short of expectations? Second, successful case studies provide evidence that the language is powerful enough to express solutions to the kinds of problems for which it was designed. I will use the case studies as opportunities to revise the language so that it is as effective as possible for users. I will also use the case studies to compare with Solidity by writing Solidity implementations of the same programs to see how they compare. Key research questions to address in case studies include:

- To what extent does the permissions system in Obsidian allow static specification of typestate, given the aliasing structure in the case? What implications does the permissions system have on the structure of the program (as compared with the structure that one might use in Solidity, for example)?
- How do the Solidity and Obsidian implementations compare? Is one implementation substantially longer? Do they expose the same interfaces to clients, or does Obsidian result in a different kind of interface?
- Of the bugs that the author of the case study initially adds, which ones (and how many) are detected by the Obsidian compiler? Are the bug patterns similar in the Solidity and the Obsidian development efforts?

Case studies do not provide quantitative evidence of quality; quantitative evidence is reserved for a comparative study (§4.4.6). Case studies are particularly useful when they are done by people other than the language designer, since other users have different perspectives to contribute. Ideas that seemed natural to the language designer may be quite foreign to others. By working with undergraduate students on the case studies, I hope to obtain evidence that Obsidian is effective for more programmers than just myself.

I have selected two case studies to do. The first one is underway; the second is an early-stage idea based on a local contact I made. However, I have several other ideas for case studies to do in case the second case study is untenable or uninformative, perhaps due to lack of engagement from my contact.

Parametric Insurance Typical insurance policies insure against particular losses, such as the loss of a house in case of fire or the loss of a car in case of a collision. These policies require an assessment of value at the time of the loss, which is a high-cost operation that depends on a mechanism by which the parties can agree on the value. In a context where there are many claims that must be paid out at once (for example, in the case of a flood, which likely causes widespread losses in a region) or in a context in which arbitration is expensive, not trustworthy, or unavailable (for example, in some developing countries), this kind of insurance can be impractical.

In contrast, parametric insurance insures against an external event (which the insuree cannot manipulate), and issues payouts that depend only on the event. For example, a farmer might insure crops against drought; a policy might say "pay $100 if the rainfall in July 2018 in ZIP code 15213 is less than one inch." If the insured condition is met, the farmer receives the payout regardless of the value of the crops that were damaged. This requires, of course, that a trusted authority provide accurate weather records. This type of insurance is commonly used in developed countries.

In developing countries, however, the parametric insurance markets are not well-developed. We propose to implement parametric insurance in Obsidian in a collaboration with World Bank. Etherisc GmbH [27], a company in Germany, is preparing to offer parametric insurance for a variety of applications via Ethereum, providing further evidence of the real-world relevance of this case study.

Call-bell system for health care Ristcall is a Pittsburgh startup that is developing a wrist-based system for patients in nursing homes. Patients can use the device to request nursing help; the system automatically senses when a nurse enters the room and records the response time. The events are currently stored in a database. Unfortunately, some prospective nursing home residents are worried about fraud: what if the database has been tampered with so that the facility can claim better call response times than they actually have? The CEO of Ristcall is interested in moving to a blockchain-based record system in order to alleviate tampering concerns, leveraaging the immutable nature of blockchain transactions.

This case study would prototype a recordkeeping system for the nurse call product. It offers some interesting contrasts with the insurance case study; the data storage is more database-oriented and the aliasing structure is likely substantially different. There may be interesting architectural questions to consider regarding the interfaces to the storage system and performance questions to consider regarding the storage itself, which will help evaluate to what extent the properties provided by Obsidian can generalize to this application. This case study is not even begun, so it is possible it will fail due to lack of support from Ristcall or that I will find that the application is not sufficiently interesting to pursue to the end.

Alternative case studies In order to mitigate the risk of the call-bell case study not being informative (for example, due to lack of commitment by Ristcall, or due to overly-simple requirements), I have identified alternatives. First, I have collaborators at IBM who are interested in supply chain software and business logistics in general; these are better-developed than the call-bell scenario and may be more interesting. However, they are also much larger problem spaces, so it is likely that it would be important to scope the case study carefully in order to obtain relevant data on Obsidian while completing the case study in a reasonable amount of time. Second, there is a collection of applications in the Elsden et al. paper [21]. Of particular interest are applications that have already been implemented on some blockchain platform, providing an independent source of comparison. For example, *Vishrambh* [35] is a blockchain application that facilitates trusted philanthropy, in which donors and beneficiaries can obtain more assurance regarding how charitable organizations spend donations. Likewise, the UN World Food Program piloted a blockchain application to track food aid [41]. Smart energy trading is another area that has received significant attention and may be

amenable to a case study.

4.4.6 Summative user studies

User studies will compare Obsidian to Solidity [24], which is the most commonly-used domain-specific language for blockchain software. In the user studies, I will recruit participants and ask them to do programming tasks in their assigned language (either Solidity or Obsidian). The overall approach to the user studies is modeled on my previous work evaluating Glacier [14].

I do not propose to define a single quantitative measure of *effectiveness*; rather, I will look at more fine-grained measures of effectiveness as described below. Overall, I hope to show that Obsidian is more effective for programmers along the *task completion* and *bug-proneness* axes. If Obsidian is also more effective in terms of *time to task completion* for some tasks, that would also be a nice result but this may not be a realistic expectation. It is likely that obtaining the additional safety properties of Obsidian may programmers some time.

Learnability

In order to study programmers using a programming language, it is unavoidable that they first learn the language. Given the niche position of Solidity, I plan to recruit participants who are experienced with object-oriented programming and teach them either Solidity or Obsidian (depending on the condition to which they have been assigned). I will take this opportunity to (a) improve the training materials to make it easier for future participants and users to learn Obsidian; (b) obtain insight about which aspects of Obsidian are particularly challenging to learn and potentially identify language changes that might address these problems. However, since learnability is not a key goal of the language, it will suffice to be able to show that programmers are *eventually* able to program effectively in Obsidian (as measured by the tasks that will follow the learning phase of the study). It will not be regarded as a failure of Obsidian if learning it takes marginally longer than learning Solidity, since Obsidian provides stronger safety guarantees.

Task completion

In this phase of the user study, I will assess task completion rates and times for small tasks that are designed to be representative of programming tasks on blockchain programs. Variance is typically very high in user studies of programming tasks, for example because users frequently get confused with issues that are irrelevant to the independent variable of interest. Keeping tasks small reduces variance by focusing the participants on the relevant parts of the problem, making it more likely that I will obtain statistically significant results. The small task size is a threat to external validity, but I will mitigate this by including several different kinds of tasks and by using case studies in addition to user studies.

It is not a requirement that Obsidian programmers necessarily complete the tasks *faster* than the Solidity programmers; it may well be that the obtaining safety guarantees or working in a more expressive language costs programmers time in the short term, especially with tasks in small programs. However, in order for Obsidian to be considered successful, it needs to be the case that programmers can actually complete their work in Obsidian, even if it takes them a little bit longer. If Obsidian were so complicated or hard to use that programmers were unable to finish reasonable tasks that are straightforward in Solidity, I would need to iterate on the training materials, programming tools, or the Obsidian language itself.

Bug-proneness

Perhaps the most important empirical evaluation question for Obsidian is whether it actually prevents real bugs. That is, do programmers using Obsidian tend to write code with fewer – or less severe – bugs than programmers using Solidity? Although it will be known from the formal work (§4.4.1) that all Obsidian programs have some specific safety properties, it might be the case that Solidity programmers do not actually make the mistakes that Obsidian detects. We already have in-the-wild evidence of certain kinds of bugs that Obsidian prevents as well as some evidence from prior studies of bug-prone aspects of Solidity [20]. However, it would be more compelling if it were known that these kinds of bugs occur *frequently* in reasonable kinds of situations. I plan to quantify to what extent the participants tend to introduce these bugs when using Solidity by giving them tasks in which they have opportunities to make mistakes that Obsidian would prevent; then, I will see what happens when Obsidian programmers work on those same tasks. Do they make the same mistakes? If so, do the error messages they obtain enable them to easily fix their bugs? This approach would show directly that Obsidian prevents bugs relative to Solidity.

4.5 Future work

For this thesis, I have proposed to compare Obsidian with Solidity, which is closest comparable blockchain programming language. However, both are object-oriented languages. Although this choice was motivated by the practical evidence that object-oriented languages have been widely used for programs that must maintain persistent state, functional approaches have been proposed to improve programmers' abilities to reason about their programs. In the future, it would be interesting to design a functional language in a similarly user-centered way, and then do a summative comparison with Obsidian. For which programmers and tasks can we show benefits of each language design? What insights can we obtain from formative user-centered methods on the design of a functional language? In doing so, I may be able to address parts of the functional vs. object-oriented language debate, which would be of much broader interest.

Chapter 5

Timeline

This is an aggressive timeline, assuming that the work proceeds quickly. It is likely that I will encounter barriers along the way. In addition, there is a possibility of spending some time consulting with IBM, which may slow the work a bit.

September 2018: Finish insurance case study, refining Obsidian implementation in the process

September–October 2018: Run qualitative pilots followed by quantitative Internet-based studies of ownership and permissions

November 2018: Prove soundness of Obsidian formalism

December 2018: Do a second case study of Obsidian, likely with a local medical devices startup

January 2019: Write and submit (to TOPLAS) a paper including the formalism and case studies

January–February 2019: Design summative study comparing Obsidian to Solidity (will likely require refining Obsidian tools further)

March 2019: Run summative study

April 2019: Analyze results from summative study

April 2019: Submit paper on preliminary and summative studies (expecting there will be methodological contributions) to ASE or OOPSLA

May–June 2019: Write thesis document

July 2019: Thesis committee reviews thesis

August 2019: Thesis defense

Bibliography

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640073. URL http://doi.acm.org/10.1145/1639950.1640073.

[2] D. Ancona, V. Bono, and M. Bravetti. *Behavioral Types in Programming Languages*. Now Publishers Inc., Hanover, MA, USA, 2016. ISBN 1680831348, 9781680831344.

[3] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts. Technical report, Cryptology ePrint Archive: Report 2016/1007, https://eprint. iacr. org/2016/1007, 2016.

[4] S. Balzer and F. Pfenning. Objects as session-typed processes. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 13–24. ACM, 2015.

[5] S. Balzer and F. Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):37, 2017.

[6] C. Barnaby, M. Coblenz, T. Etzel, E. Kanal, J. Sunshine, B. Myers, and J. Aldrich. A user study to inform the design of the obsidian blockchain dsl. In *PLATEAU '17 Workshop on Evaluation and Usability of Programming Languages and Tools*, 2017.

[7] K. Bhargavan, N. Swamy, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, and T. Sibut-Pinote. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*, pages 91–96, New York, New York, USA, 2016. ACM Press. ISBN 9781450345743. doi: 10.1145/2993600.2993611.

[8] K. Bierhoff and J. Aldrich. Plural: Checking protocol compliance under aliasing. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 971–972, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1370175.1370213. URL http://doi.acm.org/10.1145/1370175.1370213.

[9] K. Bierhoff, N. E. Beckman, and J. Aldrich. Checking concurrent typestate with access permissions in plural: A retrospective. *Engineering of Software*, pages 35–48, 2011.

[10] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40325-6. URL http://dl.acm.org/citation.cfm?id=1760267.1760273.

[11] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, pages 222–236. Springer, 2010.

[12] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring language support for immutability. In *International Conference on Software Engineering*, 2016. ISBN 978-1-4503-3900-1.

[13] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring language support for immutability. Technical Report CMU-ISR-16-106, Carnegie Mellon University, 2016.

[14] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. Glacier: Transitive class immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering - ICSE '17*, 2017.

[15] M. Coblenz, J. Aldrich, J. Sunshine, and B. Myers. Interdisciplinary programming language design. *position paper at: Dagstuhl Conference on Evidence About Programmers for Programming Language Design, Dagstuhl Seminar 18061*, February 2018.

[16] I. Corp. Core API - Hyperledger Fabric, 2018. https://openblockchain.readthedocs.io/en/latest/API/CoreAPI/#rest-api. Accessed May 29, 2018.

[17] A. Das, J. Hoffmann, and F. Pfenning. Parallel complexity analysis with temporal session types. *arXiv preprint arXiv:1804.06013*, 2018.

[18] A. Das, J. Hoffmann, and F. Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 305–314. ACM, 2018.

[19] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming: 18th European Conference, Oslo, Norway, June 14-18, 2004. Proceedings*, pages 465–490, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24851-4. doi: 10.1007/978-3-540-24851-4_21. URL https://doi.org/10.1007/978-3-540-24851-4_21.

[20] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. *IACR Cryptology ePrint Archive*, 2015:460, 2015.

[21] C. Elsden, A. Manohar, J. Briggs, M. Harding, C. Speed, and J. Vines. Making sense of blockchain applications: A typology for hci. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 458:1–458:14, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5620-6. doi: 10.1145/3173574.3174032. URL http://doi.acm.org/10.1145/3173574.3174032.

[22] Ethereum Foundation. Ethereum project, 2017. http://www.ethereum.org. Accessed Jan. 3, 2017.

[23] Ethereum Foundation. Common patterns, 2017. URL http://solidity.readthedocs.io/en/develop/common-patterns.html.

[24] Ethereum Foundation. Solidity. https://solidity.readthedocs.io/en/develop/, 2018. Accessed May 16, 2018.

[25] E. Foundation. Vyper, 2018. URL https://vyper.readthedocs.io/en/latest/.

[26] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, Oct. 2014. ISSN 0164-0925. doi: 10.1145/2629609. URL http://doi.acm.org/10.1145/2629609.

[27] E. GmbH. Etherisc - decentralized insurance, 2018. URL https://etherisc.com.

[28] R. Graham. Ethereum/thedao hack simplified, 2016. URL http://blog.erratasec.com/2016/06/etheriumdao-hack-similfied.html#.WgCuCrbGy3J.

[29] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, oct 2014. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-013-9289-1.

[30] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578. ACM, 2009.

[31] S. Kurtev, T. A. Christensen, and B. Thomsen. Discount method for programming language evaluation. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2016, pages 1–8, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4638-2. doi: 10.1145/3001878.3001879. URL http://doi.acm.org/10.1145/3001878.3001879.

[32] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016. ISBN 9781450341394. doi: 10.1145/2976749.2978309.

[33] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[34] B. Marino. Smart-contract escape hatches: The dao of the deo. http://hackingdistributed.com/2016/06/22/smart-contract-escape-hatches/, 2016. Accessed June 5, 2018.

[35] A. Mehra and S. Lokam. Vishrambh: Trusted philanthropy with end-to-end transparency. *HCI for Blockchain: A CHI 2018 workshop on Studying, Critiquing, Designing and Envisioning Distributed Ledger Technologie*, 2018.

[36] Mozilla Research. The Rust programming language. https://www.rust-lang.org, 2016. Accessed Feb. 8, 2016.

[37] B. A. Myers, J. F. Pane, and A. Ko. Natural programming languages and environments. *Communications of the ACM*, 47:47–52, 2004.

[38] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. *ACM SIGPLAN Notices*, 47(1):557–570, 2012.

[39] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM, 1990.

[40] J. F. Pane, B. A. Myers, and L. B. Miller. Using hci techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 198–206. IEEE, 2002.

[41] W. F. Programme. Building blocks, 2018. URL http://innovation.wfp.org/project/building-blocks.

[42] A. Project. Introduction to ergo, 2018. URL https://docs.accordproject.org/docs/ergo.html.

[43] R3. Corda: the open source blockchain for business, 2018. URL https://www.corda.net.

[44] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in flint. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, pages 218–219, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5513-1. doi: 10.1145/3191697.3213790. URL http://doi.acm.org/10.1145/3191697.3213790.

[45] A. Stefik and S. Hanenberg. The programming language wars: Questions and responsibilities for the programming language community. In *Proceedings of the 2014 ACM International Symposium on New*

Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, pages 283–299, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661156. URL http://doi.acm.org/10.1145/2661136.2661156.

[46] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, Jan 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312929.

[47] J. Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, USA, 12 2013. CMU-ISR-13-117.

[48] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter. First-class state change in Plaid. In *ACM SIGPLAN Notices*, pages 713–732. ACM, 2011.

[49] J. Sunshine, J. D. Herbsleb, and J. Aldrich. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[50] J. Sunshine, J. D. Herbsleb, and J. Aldrich. Searching the state space: A qualitative study of api protocol usability. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 82–93, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820295.

[51] N. Szabo. Formalizing and securing relationships on public networks, 1997.

[52] The Linux Foundation. Hyperledger, 2017. https://www.hyperledger.org. Accessed Jan. 3, 2017.

[53] J. A. Tov and R. Pucella. Practical affine types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926436. URL http://doi.acm.org/10.1145/1926385.1926436.

[54] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359, 1990.

[55] M. Willsey, R. Prabhu, and F. Pfenning. Design and implementation of concurrent c0. *arXiv preprint arXiv:1701.04929*, 2017.

[56] P. T. Wilson, J. Pombrio, and S. Krishnamurthi. Can we crowdsource language design? In *Symposium on New Ideas in Programming and Reflections on Software*, Onward! 2017, 2017.