# Obsidian: Typestate and Assets for Safer Smart Contracts

ANONYMOUS AUTHOR(S)

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

Additional Key Words and Phrases: programming language design, smart contract programming languages, blockchain, aliasing, permissions systems, usability of programming languages

## 1 INTRODUCTION

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

Proponents have suggested that blockchains be used for a plethora of applications, such as finance, health care [Harvard Business Review 2017], supply chain management [IBM 2019], and others [Elsden et al. 2018]. Unfortunately, some prominent blockchain applications have included security vulnerabilities, for example through which over $80 million worth of virtual currency was stolen [Graham 2017; Sirer 2016]. In addition to the potentially severe consequences of bugs, platforms require that contracts are immutable, so bugs cannot be fixed easily. If organizations are to adopt blockchain environments for business-critical applications, there needs to be a more reliable way of writing smart contracts.

Many techniques promote program correctness, but our focus is on programming language design so that we can prevent bugs as early as possible — potentially by aiding the programmer's reasoning processes before code is even written. We have created *Obsidian*, 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* to statically ensure that objects are manipulated correctly according to their current states, and uses *linear types* to enable safe manipulation of assets, which must not be accidentally lost.

Obsidian's sophisticated features for improving safety will not do any good if programmers cannot or will not use them. Therefore, we aim to provide *usable* features: ones that are designed so that people can use them effectively. Although techniques for developing programming languages in a human-centered way are not yet mature, one of our research goals in developing Obsidian is to develop principles and techniques for designing languages that are *effective* for their users. However,

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

it does not suffice to focus entirely on usability; the language must provide strong, domain-relevant guarantees. We rely on theory to confine our design to the space of languages that have the formal properties we desire, permitting safe, sound reasoning about program properties.

Following Coblenz et al. [2018], we have adapted methods from the human-computer interaction literature to make it more likely that Obsidian will be a practical, effective language for programmers to use. For example, we use some sophisticated type system ideas from the programming language community that have not yet been adopted in practice, which creates a usability risk. Therefore, we used human-centered techniques to help refine those ideas to increase their adoptability and utility. These methods are *formative* — that is, they serve to provide insight in the early stages of the design process. Formative methods contrast with *summative* methods, which are used to evaluate a completed design (usually in a quantitative way). We describe several results gathered from these techniques that influenced our design.

Analyzing usability requires first deciding who the users are. We take the perspective that we want to empower as many people as possible to write or modify Obsidian programs, while accepting that prior programming expertise will be required for many kinds of tasks.

We make the following contributions:

(1) We show how typestate and linear types can be combined in a programming language, using a rich but simple permission system that captures the required restrictions on aliases.
(2) We show how low-cost user studies can be used to obtain insights, which we leveraged to refine the programming language design.
(3) As case studies, we show how Obsidian can be used to implement a parametric insurance application and a supply chain, and by comparison to Solidity, how leveraging typestate can move checks from run time to compile time. Our case studies were done by programmers who were not the designers of the language, showing that the language is usable by people other than only the designers.

After summarizing related work, we introduce the Obsidian language with an example (§3). We describe how the language design fits into the Fabric blockchain infrastructure in §4. Section 5 focuses on the design of particular aspects of the language and describes how qualitative studies influenced our design. We discuss two case studies in §6. Future work is discussed in §7. We conclude in §8.

## 2 RELATED WORK

Researchers have previously investigated common causes of bugs in smart contracts [Atzei et al. 2016; Delmolino et al. 2015; Luu et al. 2016], created static analyses for existing languages [Kalra et al. 2018], and worked on applying formal verification [Bhargavan et al. 2016]. Our work focuses on preventing bugs in the first place by designing languages in which many commonplace bugs cannot occur.

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

Some researchers have used quantitative studies to consider specific design questions, such as static vs. dynamic types [Hanenberg et al. 2014]. Quantitative studies are useful at the conclusion of a design process, but are too resource-intensive to apply for each individual design decision along the way.

Our work instead uses *qualitative* studies to focus on exploring design alternatives within a design space that theory suggests may be good. Prior qualitative work on usable programming languages includes Kurtev et al.'s study on Quorum [Kurtev et al. 2016] 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 [Wilson et al. 2017]. Neither of these studies consider typestate or linearity features.

DeLine investigated using typestate in the context of object-oriented systems [DeLine and Fähndrich 2004], finding that subclassing causes complicated issues of partial state changes; we avoid that problem by not supporting subclassing. Plaid [Sunshine et al. 2011] and Plural [Bierhoff and Aldrich 2008] are the most closely-related systems in terms of their type systems' features, but neither language used formative studies to inform 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 [Sunshine et al. 2014], but that study (and others on typestate) did not ask participants to write code that used typestate. The designers of Plural performed summative case studies themselves, but did not invite participants for *formative* user studies [Bierhoff et al. 2011]. Our work rests in part on the theoretical foundations of typestate in Garcia et al. [2014].

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

## 3 INTRODUCTION TO THE OBSIDIAN LANGUAGE

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

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

We were particularly interested in creating a language that we would eventually be able to evaluate with users, while at the same time significantly improving safety relative to existing

language designs. In short, we aimed to create a language that we could show was more effective for programmers. In order to make this practical, we made some relatively standard surface-level design choices that would enable our users to learn the core language concepts more easily, while using a sophisticated type system to provide strong guarantees. Where possible, we chose approaches that would enable static enforcement of safety, but in a few cases we moved checks to runtime in order to enable a simple design for users or a more precise analysis (for example, in dynamic state checks, §5.6).

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* [Mikhajlov and Sekerinski 1998]). 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.

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

Smart contracts commonly manipulate *assets*, such as virtual currencies. Some common smart contract bugs pertain to accidental loss of assets [Delmolino et al. 2015]. If a contract 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.

To enforce this, 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 in the syntax 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 `Wallet` object can be written `Wallet@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 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 an object, it may have `Shared` and `Unowned` aliases. Examples of these scenarios are shown in Fig. 2 to provide some intuition.

In Obsidian, the *mode* portion of a type can change due to operations on a reference, so transaction signatures can specify modes both before and after execution. As in Java, a first argument called `this` is optional; when present, it is used to specify initial and final modes on the receiver. The ≫ symbol separates the initial mode from the final one. In the example above, `buy` must be invoked

```
1  // This vending machine sells candy in exchange for candy tokens.
2  main asset contract TinyVendingMachine {
3      Coins @ Owned coinBin;
4
5      state Full {
6          Candy @ Owned inventory;
7      }
8      state Empty; // No candy if the machine is empty.
9
10     TinyVendingMachine() {
11         coinBin = new Coins(); // Start with an empty coin bin.
12         ->Empty;
13     }
14
15     transaction restock(TinyVendingMachine @ Empty >> Full this,
16                         Candy @ Owned >> Unowned c) {
17         ->Full(inventory = c);
18     }
19
20     transaction buy(TinyVendingMachine @ Full >> Empty this,
21                     Coin @ Owned >> Unowned c) returns Candy @ Owned {
22         coinBin.deposit(c);
23         Candy result = inventory;
24         ->Empty;
25         return result;
26     }
27
28     transaction withdrawCoins() returns Coins @ Owned {
29         Coins result = coinBin;
30         coinBin = new Coins();
31         return result;
32     }
33 }
```

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

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 10 above), which initialize fields as needed. If a contract has any states declared, then every instance of the contract must be in one of those states from the time each constructor exits.

Objects in smart contracts frequently maintain high-level state information [Ethereum Foundation 2017], with the set of permitted transactions depending on the current state. For example, a `TinyVendingMachine` might be `Empty` or `Full`, and the buy transaction can only be invoked on a `Full` machine. Prior work showed that including state information in documentation helped users understand how to use object protocols [Sunshine et al. 2014], so we include first-class support for states in Obsidian. *Typestate* [Aldrich et al. 2009] is the idea of including state information in types, and we take that approach in Obsidian so that the compiler can ensure that objects are manipulated correctly according to their states. State information can be captured in a

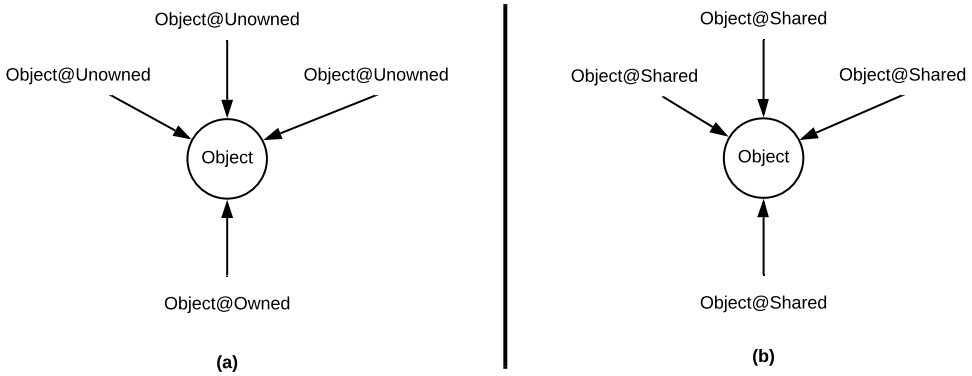


Fig. 2. Some common aliasing scenarios. (a) shows an object with one owner; (b) shows a shared object.

mode. For example, `TinyVendingMachine@Full` is the type of a reference to an object of contract `TinyVendingMachine` with mode `Full`. In this case, the mode denotes that the referenced object is statically known to be in state `Full`.

State is mutable; objects can transition from their current state to another state via a transition operation. For example, `->Full(inventory = c)` might change the state of a `TinyVendingMachine` to the `Full` state, initializing the `inventory` field of the `Full` state to `c`. This leads to a potential difficulty: what if a reference to a `TinyVendingMachine` with mode `Empty` exists while the state transitions to `Full`? To prevent this problem, typestate is only available with references that also have ownership. Because of this, there is no need to separately denote ownership in the syntax; we simply observe that every typestate-bearing reference is also owned. Then, Obsidian restricts the operations that can be performed through a reference according to the reference's mode. In particular, if an owned reference might exist, then non-owning references cannot be used to mutate typestate. If no owned references exist, then all references permit state mutation. A summary of modes is shown in Table 1.

| **Mode** | **Meaning** | **Restrictions** |
|---|---|---|
| `Owned` | This is the only reference to the object that is owned. There may be many `Unowned` aliases but no `Shared` aliases. | Typestate mutation permitted |
| `Unowned` | There may or may not be any owned aliases to this object, but there may be many other `Unowned` or `Shared` aliases. | Typestate mutation forbidden |
| `Shared` | This is one of potentially many `Shared` references to the object. There are no owned aliases. | Typestate mutation 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. | Typestate mutation permitted |

Table 1. A summary of modes in Obsidian

## 4 SYSTEM DESIGN AND IMPLEMENTATION

Obsidian supports Hyperledger Fabric [The Linux Foundation 2018], 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. 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. 4.

### 4.1 Storage in the ledger

Fabric provides a key/value store for persisting the state of smart contracts in the ledger. As a result, Fabric requires that smart contracts serialize their state in terms of key/value pairs. In other smart contract languages, programmers are required to manually write code to serialize and deserialize their smart contract data. In contrast, Obsidian automatically generates serialization code, leveraging *protocol buffers* [Google Inc. 2019] to map between message formats and sequences of bytes. When a transaction is executed, the appropriate objects are lazily loaded from the key/value store as required for the transaction's execution. Lazy loading is *shallow*: the object's fields are 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 made is very cheap, since this entails *not* setting key/value pairs in the store, flushing the heap of objects that have been lazily loaded, and (shallowly) re-loading the root object from the ledger. This lazy approach decreases execution cost and frees the programmer from needing to manually load and unload key/value pairs from the ledger, as would normally be required on Fabric.

### 4.2 Passing objects

In Ethereum, transaction arguments and outputs must be primitives, not objects. As a result of automatic serialization and deserialization, Obsidian permits arbitrary objects to be passed as arguments and returned from transactions. Obsidian accepts objects encoded in their protocol buffer encodings. Since the protocol buffer specifications are emitted by the Obsidian compiler, any client (even non-Obsidian clients) can correctly serialize and deserialize native Obsidian objects in the format Obsidian expects to invoke Obsidian transactions and interpret their results.

Every Obsidian object has a unique ID, and references to objects can be transmitted between clients and the blockchain via its 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 a ID factory. 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.

Although serializing objects according to their Protobuf specifications is better than requiring programmers to manually write their own serialization code, if a client is written in a traditional language, the client does not obtain the safety benefits of the Obsidian type system. Obsidian

```
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 }
```

Fig. 3. 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. 1) to have Shared receivers, since top-level objects are never owned by clients.

addresses this problem in two ways. First, clients can also be written using the Obsidian language and compiler; Obsidian clients obtain all the benefits of the Obsidian type system. The compiler can output Java code that runs as its own process (not on a blockchain) and invokes blockchain transactions remotely. 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. 3 shows a simple client program that uses the `TinyVendingMachine` above. The `main` transaction takes a `remote` reference to the smart contract instance.

### 4.3 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

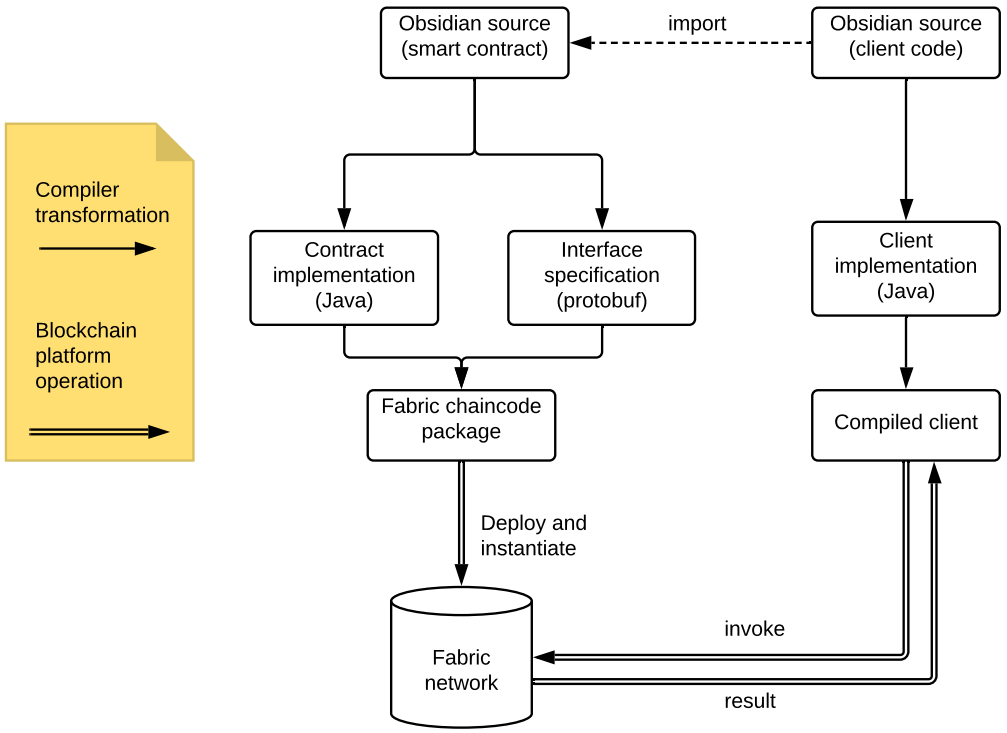


Fig. 4. Obsidian system architecture

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.

## 5 LANGUAGE DESIGN PROCESS AND DETAILS

Obsidian is the first object-oriented language (of which we are aware) to integrate linear assets and typestate. This combination — and, in fact, even just including typestate — could result in a design that was hard to use, since typical typestate languages require users to understand a complex permissions model. In designing the language, we focused on *simplicity* in service of usability. We maintained static safety where possible, but moved certain checks to runtime where needed to maintain a high level of expressiveness. We also aimed to simplify the job of the programmer relative to existing blockchain programming languages by eliminating onerous, error-prone programming tasks, such as writing serialization and deserialization code, as discussed above. In this section, we describe how we designed features to improve user experience, in some cases driven by results of

formative user studies. Rather than relying only on our own experience and intuition, we invited participants to help us assess the tradeoffs of different design options. This enabled us to take a more data-driven approach in our language design, like Stefik and Hanenberg [2014] and Coblenz et al. [2018]. We take the perspective that we should integrate *qualitative* methods in addition to quantitative methods in order to drive language design in a direction that is more likely to be beneficial for users.

### 5.1 Qualitative Studies

In designing Obsidian, we integrated several qualitative methods in our formative user studies. These studies are not the focus of this paper, but we found them useful in iterating on our design, and we hope that others will be able to leverage these techniques in the future. We obtained IRB approval for our studies and paid participants $10/hour for their time. Overall, we recruited 28 participants in the formative studies across various parts of the Obsidian design. In this paper, we give participants arbitrary identifiers, such as *P14*, to conceal their identities. Participants were generally local students studying computer science, mostly at the graduate level. Although recruiting only students would be problematic for a quantitative study in which we attempted to reason generally about programmers, in these qualitative studies, we were interested in identifying common barriers to success. Even if the difficulties encountered by one population may not be the same as those encountered by another population, we view all potential users as being valuable sources of insight regarding our design. This is a typical approach in the HCI community [Nielsen 1993]. Our goal in qualitative studies was not to gather quantitative information about the frequency of usability problems; instead, we assume (as is typical in usability studies) that any problem we see is likely to occur with many participants and is therefore worth fixing if possible.

There is a difficult practical problem running lab studies on a new programming language: if the language is very different from those with which a participant is familiar, there is a learning stage where the participant must learn the new language. One approach is to focus on language *learnability* [Stefik et al. 2011]: how easily can novices learn a language, and how well do those novices perform? Our interest, however, is in long-term programmer performance, and we aim to gather as much relevant data as possible in a short lab session. This would seem to limit our study to languages that either participants already know or which they can learn quickly.

A corresponding problem is one of conflation. Suppose one is interested in analyzing the usability of a particular design decision. If one teaches participants a new language, the participants are likely to have difficulty with many different aspects of the language, not just the one of interest. Furthermore, when a participant is confused about something related to the design decision, it is not clear whether the root cause is the design decision itself or some unrelated point of confusion or aspect of the language. The effects of the design decisions become overwhelmed by the noise from unrelated parts of the language.

We observe, however, that languages (including Obsidian) are designed so that features are as orthogonal as possible [Pratt and Zelkowitz 1996; Sebesta 2006]. Therefore, our approach is to study the design decisions in isolation by *back-porting them to a language with which participants are already familiar*. This approach introduces its own noise: perhaps the aspects do not behave in the second language as they would in the first, and the outcome might not be applicable to the first language. However, we limit the impact of any differences by choosing Java, which is structurally similar to Obsidian, and choosing tasks where the differences are not particularly important. For example, both Java and Obsidian are statically-typed object-oriented languages, and the problem of aliases to mutable state is common to both.

### 5.2 Type declarations, annotations, and static assertions

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

To alleviate this problem, we introduced *static assertions*. These have the syntax `[e @ mode]`. For example, `[account @ Open]` statically asserts that the reference `account` is owned and refers to an object that the compiler has proved 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 runtime 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.

In a user study of an earlier version of the language, participants were unsure when ownership was transferred when calling a method. In *that* version of the language, a `deposit` transaction signature might be written:

```
transaction deposit (Money @ Owned money) { ... }
```

In the final version of the language, the same signature would be written:

```
transaction deposit (Money @ Owned >> Unowned money) { ... }
```

In the earlier version of the language, the programmer was expected to understand that to pass an `Owned` reference, the caller must give up ownership. In contrast, in `logAmount`, the caller would not transfer ownership, because the parameter was `Unowned`:

```
transaction logAmount (Money @ Unowned money) { ... }
```

This distinction was too subtle. For example, in one formative study, P19 asked what happens when passing an `@Owned` object to a method with an unowned formal parameter. We regard the fact that the question was asked as an indication that the syntax is unclear. P20 said, "I am confused between when you write `@Owned` to make a variable to be owned or non-owned and the transfer of ownership. So when I [annotate this constructor type `@Owned`], I'm not sure if I'm making a variable owned or I'm transferring ownership." To address these problems, we made ownership transfer explicit in signatures of transactions: when ownership changes, both initial and final modes are written.

```
transaction deposit (Money @ Owned >> Unowned money) { ... }
```

### 5.3 State transitions

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

- When an object is in a particular state, the fields for that state should be initialized.
- When an object is *not* in a particular state, the fields for that state should be out of scope.
- According to the *user control and freedom* heuristic [Nielsen and Molich 1990], 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.

We assessed the usability of several options in a formative user study. Because this study was intended to help assess which of several options was likely best, rather than to gather statistically significant results, it was limited in size. First, we gave the four participants in the part of the study a state transition diagram and code partially implementing a `Wallet` object, and asked them to invent code to do state transitions. Notably, they all initialized fields *before* transitioning. We gave them four approaches, and asked them to use each one in a partially-completed transaction.

(1) Assets must be assigned to fields in the transition, e.g. `->S(x = a1)` indicates assigning the value of the local variable `a1` to field `x` of state `S`.
(2) Assets must be assigned to fields *before* the transition, e.g. `S::x = a1; ->S`.
(3) Assets must be assigned to fields *before* the transition, but the fields are in local scope, not in destination-state scope, e.g. `x = a1; ->S`.
(4) Assets must be assigned to fields *after* the transition, e.g. `->S; x = a1`.

No participant had significant confusion or made significant mistakes when implementing each initialization approach. Most of the participants preferred assigning assets to fields *before* the transition with destination state scoping (option 2). Most of the participants disliked assignment to fields after the transition because it conflicts with their interpretation of the semantics of state transitions. This was consistent with their behavior before being shown the options, so we decided to design Obsidian to allow this. Obsidian supports both options (1) and (2) above. This is consistent with all three of our objectives above: users can initialize fields before transitions or during transitions, but without needing access to fields that are out of scope given the current state and without ever having uninitialized current-state fields.

### 5.4 Transaction scope

Since some transactions are only available when the object is in a particular state, some previous typestate-oriented languages supported defining methods inside states. For example, Plaid [Sunshine et al. 2011] allows users to define the `read` method inside the `OpenFile` state to make clear that `read` can only be invoked when a `File` is in the `OpenFile` state. Barnaby et al. [2017] considered this question and observed that study participants, who were given a typestate-oriented language that included methods in states, asked a lot of 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`.

### 5.5 Field type consistency

In traditional object-oriented languages, fields always reference either `null` or 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`? An obvious way would be as follows:

```
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  }
```

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:

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

Although this approach might seem like a reasonable consequence of the desire to keep field values consistent with their types, it imposes a significant burden. First, the programmer is required to introduce additional states, which leaks implementation details into the interface. Second, this requires that transitions return the newly out-of-scope fields, but it is not clear how: should the result be of record type? Should it be a tuple? What if the programmer neglects to do something with the result? Plaid [Sunshine et al. 2011] addressed the problem by not including type names in fields, but that approach may hamper code understandability [Coblenz et al. 2014]. In Obsidian, we permit fields to *temporarily* reference objects that are not consistent with the fields' declarations, but we require that at the end of methods (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 must always refer to instances of appropriate contracts; the only discrepancy permitted is of mode. The same is true for transaction parameters, except that the type of a parameter at the end of the method must

be a subtype of the specified final type. Obsidian forbids re-assigning formal parameters to refer to other objects to ensure soundness of this analysis.

Re-entrancy imposes a significant problem here: 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. To address this, Obsidian forbids re-entrant calls to *public* methods at the object level of granularity. The Obsidian runtime detects illegal re-entrant calls and aborts transactions that attempt them. However, to facilitate helper methods, Obsidian also supports `private` methods, which declare the expected types of the fields before and after the invocation. For example:

```
contract PrivateTransactions {
    C @ S1 c;
    private (C @ S2 >> S1 c) transaction t1() {...}
}
```

Transaction `t1` may only be invoked by methods of `PrivateTransactions`, and only on `this`. When `t1` is invoked, the typechecker checks to make sure field `c` has type `C @ S2`, and assumes that after `t1` returns, `c` will have type `C @ S1`.

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

### 5.6 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 show this statically. For example, consider `redeem` in Fig. 5. After line 24, the contract may be in either state `Active` or state `Expired`. However, inside the dynamic state check block that starts on line 26, 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:

- 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 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 runtime, if any *other* `Shared` reference is used to change the state of the referenced object, then the transaction is aborted. 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 runtime 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 typechecker expected, but we expect our approach has significantly lower runtime cost. Furthermore, our approach results in errors occurring immediately on transition, not later on in execution, 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 as usual. 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.

We conducted user studies on the usability of a prototype Obsidian permissions system. Although those user studies focused on the *static* aspects of the system, we observed that several of the participants found it more natural to think of state in a *dynamic* way. That is, rather than writing down type signatures that specified types, they wrote tests using `if` to check state. We concluded that as programmers start using Obsidian and learn how to reason about its type system, they are likely to add dynamic state checks in cases where a static design might be more appropriate.

The dynamic state check mechanism is related to the *focusing* mechanism of Fahndrich and DeLine [2002]. Unlike focusing, Obsidian's dynamic state checks detect unsafe uses of aliases precisely rather than conservatively, 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.

## 6 EVALUATION

Beyond the formative user studies that helped us design the language, 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, which are a typical way of evaluating new programming languages [cite][cite].

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:

(1) Does the aliasing structure in real blockchain applications allow use of ownership (and therefore typestate)? 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?
(2) How does the design of Obsidian impact the architecture of smart contracts?
(3) To what extent does the use of typestate reduce the need for explicit state checks and assertions, which would otherwise be necessary?

### 6.1 Case study 1: Parametric Insurance

*6.1.1 Motivation.* To address the research questions above, we were interested in implementing a blockchain application in Obsidian. To obtain realistic results, we looked for a domain in which:

- Use of a blockchain platform for the application provided significant advantages over a traditional, centralized platform.

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

Fig. 5. A dynamic state check example.

- 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.

A summary of this case study based on an earlier version of the language was previously described by Koronkevich [2018][1], but here we provide substantially more analysis and the implementation is more complete.

In *parametric insurance*, a buyer purchases a claim, specifying a *parameter* that governs when the policy will pay out. For example, a farmer might buy drought insurance as parametric insurance, specifying that if the soil moisture index 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 both 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 workflow for participants.

*6.1.2 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 when appropriate.
- 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.

*6.1.3 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.

[1]Unpublished draft.

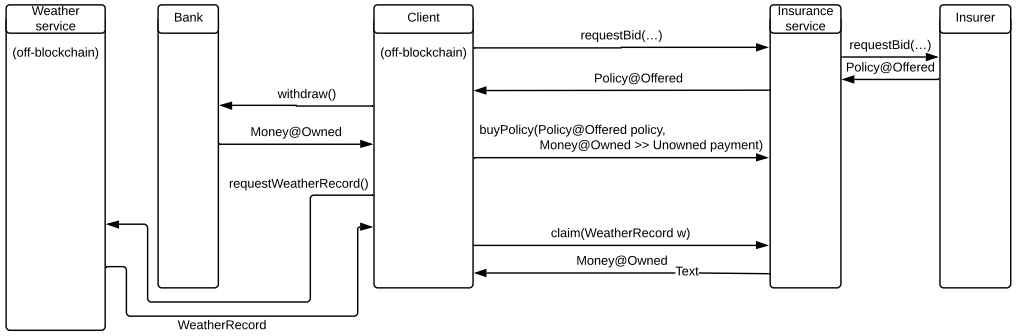


Fig. 6. Invocations sent and results returned in a typical successful bid/claim scenario.

- 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.

*6.1.4 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. 6. All of the objects reside in the blockchain except as noted. The full code for this case study is in the supplemental materials.

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 application design and architecture with Obsidian and other typestate-oriented programming languages.

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 `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` was 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` applies to individual states rather than only 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. A significant line of research has focused on *ownership types* [Clarke et al. 1998], which refers to a different notion of ownership than we use here in Obsidian. Ownership types aim to enforce encapsulation by ensuring that the implementation of an object cannot leak outside its owner. Here, we are less concerned with encapsulation and more focused on sound typestate semantics. This allows us to avoid the strict nature of these encapsulation-based approaches while accepting their premise: typically, good architecture results in an aliasing structure in which one "owner" of a particular object controls the object's lifetime and, likely, all of the changes to the object. UML distinguishes between composition, which implies ownership, and aggregation, which does not, reinforcing the idea that ownership 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 the case study, we found this 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. Then, the insurance service must associate the currency for a policy with the policy itself. Does the policy, then, 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 in this strict way 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 Solidity implementation, which we wrote for comparison purposes. Figure 7 shows how the Obsidian implementation is 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.

```
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 _expirationTime) public {
    cost = _cost;
    expirationTime = _expirationTime;
    currentState = States.Offered;
  }

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

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

(b) Solidity implementation of a Policy contract.

Fig. 7. Comparison between Obsidian and Solidity implementations of a Policy contract from the insurance case study.

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.

## 6.2 Case study 2: Shipping

*6.2.1 Motivation.* Supply chain tracking is one of the commonly-proposed applications for blockchains [IBM 2019]. As such, we were interested in what implications Obsidian's design would have on an application that tracks shipments as they move through a supply chain. We collaborated with partners in an industrial research organization[2] to conduct a case study of a simple shipping application. Our collaborators wrote most of the code, with occasional Obsidian help from us.

*6.2.2 Results.* The implementation consists of 367 non-comment, non-whitespace lines of Obsidian code. (The full implementation is included in the supplemental materials.) We found it very encouraging that they were able to write the case study with relatively little input from us, which is remarkable considering that Obsidian is a research prototype with extremely limited

[2]Their identities can be revealed in the non-blind version of this paper.

documentation. Although this is smaller than the insurance case study, we noticed some interesting relationships between the Obsidian type system and object-oriented design.

Fig. 8 summarizes an early design of the Shipping application, focusing on a particular ownership problem. 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` method attempts to append a new `Leg` to its `legList`. It does so by calling the `Leg` constructor, which takes a `Shared Transport`. But calling this constructor passing an owned reference (`this`) causes the caller's reference to become `Shared`, not Owned, which is inconsistent with the type of `depart`, which requires that `this` be owned (and specifically in state `InTransport`).

Fig. 9 shows the final design of the application. This version passes the type checker. Note how a `LegList` contains only `Arrived` references to `Leg` objects. One `Leg` may be `InTransit`, but that is owned by the `Transport` when it is in an appropriate state (also `InTransit`). 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. 8), `LegList` is responsible for both maintaining the list of legs as well as recording when the first leg arrived. This violates the *single responsibility principle* [Martin et al. 2003]. In the revised version, `LegList` only maintains a list of `Leg` objects; updating their states is implemented elsewhere.

One difficulty we noticed in this case study, however, is that sometimes there is a conceptual gap between the relatively low-level error messages given by the compiler and the high-level design changes needed in order to improve the design. For example, the first error message in the initial version of the application shown in Fig. 8 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.

## 7 FUTURE WORK

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

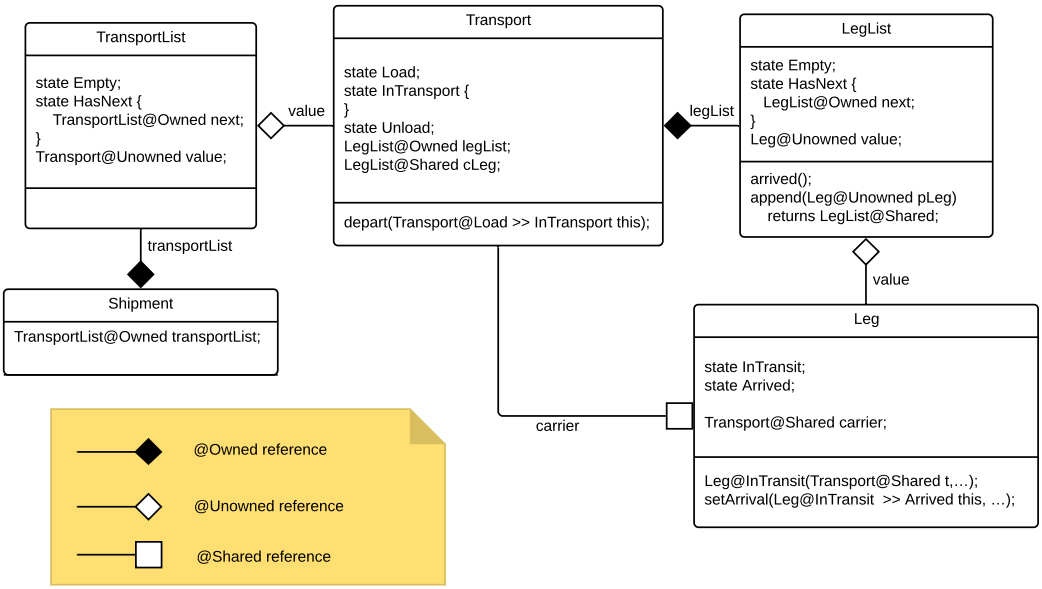


Fig. 8. Initial design of the Shipping application.

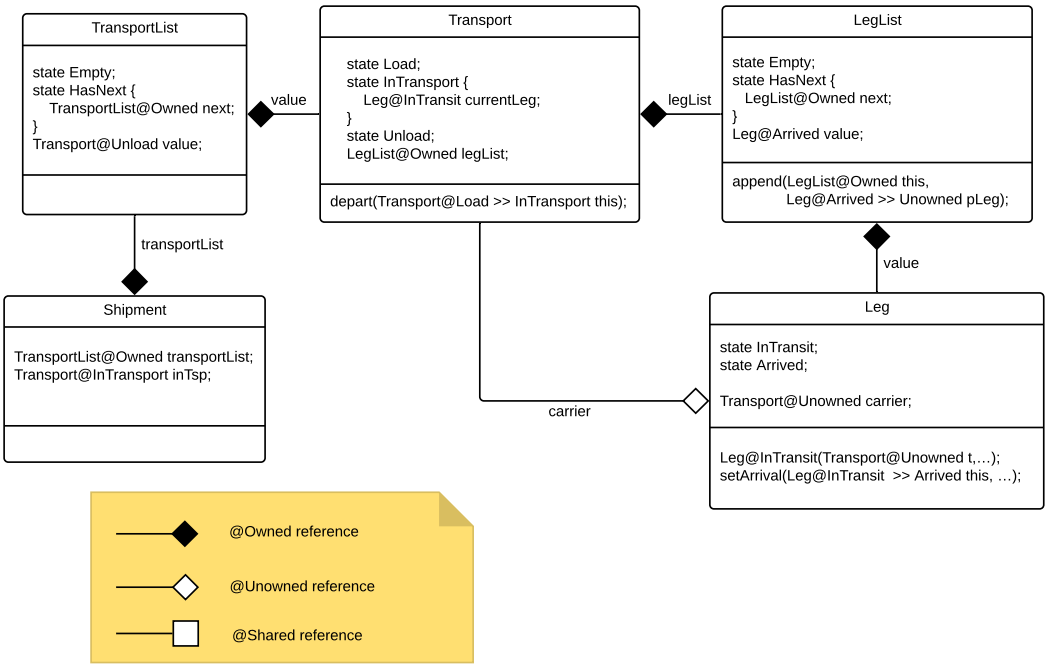


Fig. 9. Revised design of the Shipping application.

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

Obsidian currently has no IDE support; we plan to develop plugins for a popular IDE so that programmers can edit Obsidian code more conveniently and efficiently.

In the current implementation, Obsidian clients invoke all remote transactions sequentially. This means that another remote user might run intervening transactions, violating assumptions of the client program. We plan to address this by using a Fabric mechanism to group transactions into larger Fabric transactions, which will fail if any conflicting transaction occurred.

The type system-oriented approach in Obsidian is beneficial for many users, but it does not lead to verification of domain-specific program properties. In the future, it would be beneficial to augment Obsidian with a verification mechanism so that users can prove relevant properties of their programs formally.

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

## 8 CONCLUSIONS

With Obsidian we have shown how:

- Typestate can be combined with assets using a simple permissions system to provide relevant safety properties for smart contracts
- Qualitative user studies can be integrated into programming language design to lead to useful design insights
- Simple applications can be built successfully with typestate and assets, with useful implications on architecture and object-oriented design

Obsidian represents a promising direction in the design of smart contract languages and programming languages in general. We expect that the qualitative research methods will enjoy further adoption in designing programming languages for other domains, and that the innovations in the Obsidian type system will find use outside blockchain systems.

## REFERENCES

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. *A survey of attacks on Ethereum smart contracts.* Technical Report. Cryptology ePrint Archive: Report 2016/1007, https://eprint. iacr. org/2016/1007.

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 *PLATEAU '17 Workshop on Evaluation and Usability of Programming Languages and Tools.*

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 *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. New York, New York, USA. https://doi.org/10.1145/2993600.2993611

Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: Checking Protocol Compliance Under Aliasing. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 971–972. https://doi.org/10.1145/1370175.1370213

Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. 2011. Checking concurrent typestate with access permissions in plural: A retrospective. *Engineering of Software* (2011), 35–48.

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*. Springer, 222–236.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2014. Considering Productivity Effects of Explicit Type Declarations. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '14)*. ACM, New York, NY, USA, 59–61. https://doi.org/10.1145/2688204.2688218

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Onward! 2018 Essays (SPLASH '18)*.

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019. Smarter Smart Contract Development Tools. *Proceedings of 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (2019).

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

Robert DeLine and Manuel Fähndrich. 2004. *Typestates for Objects*. Springer Berlin Heidelberg, Berlin, Heidelberg, 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

Kevin Delmolino, Mitchell Arnett, Ahmed E Kosba, Andrew Miller, and Elaine Shi. 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive* 2015 (2015), 460.

Chris Elsden, Arthi Manohar, Jo Briggs, Mike Harding, Chris Speed, and John Vines. 2018. Making Sense of Blockchain Applications: A Typology for HCI. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 458, 14 pages. https://doi.org/10.1145/3173574.3174032

Ethereum Foundation. 2017. Common Patterns. (2017). Retrieved November 6, 2017 from http://solidity.readthedocs.io/en/develop/common-patterns.html

Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/512529.512532

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. https://doi.org/10.1145/2629609

Google Inc. 2019. Protocol Buffers. https://developers.google.com/protocol-buffers/

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

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

Harvard Business Review. 2017. The Potential for Blockchain to Transform Electronic Health Records. (2017). Retrieved October 31, 2017 from https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records

Dominik Harz and William Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. arXiv:cs.CR/1809.09805

IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from https://www.ibm.com/blockchain/supply-chain/

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. NDSS.

Paulette Koronkevich. 2018. Obsidian in the Rough: A Case Study Evaluation of a New Blockchain Programming Language. In *SPLASH Student Research Companion 2018 (SPLASH '18)*. (unpublished manuscript).

Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. 2016. Discount Method for Programming Language Evaluation. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3001878.3001879

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of ACM CCS'16*. https://doi.org/10.1145/2976749.2978309

R.C. Martin, J.M. Rabaey, A.P. Chandrakasan, and B. Nikolic. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education. https://books.google.com/books?id=0HYhAQAAIAAJ

Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, UK, 355–382.

Jakob Nielsen. 1993. *Usability engineering*. Academic Press, Boston.

Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 249–256.

Terrence W. Pratt and Marvin V. Zelkowitz. 1996. *Programming Languages: Design and Implementation*.

Mozilla Research. 2015. The Rust Programming Language. (2015). https://www.rust-lang.org. Accessed Feb. 8, 2016. First stable release in 2015.

Franklin Schrans and Susan Eisenbach. 2019. Introduce the Asset trait. https://github.com/flintlang/flint/blob/master/proposals/0001-asset-trait.md

Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing Safe Smart Contracts in Flint. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*. ACM, New York, NY, USA, 218–219. https://doi.org/10.1145/3191697.3213790

Robert W. Sebesta. 2006. *Concepts of Programming Languages, Seventh Edition.*

Emin Gün Sirer. 2016. Thoughts on The DAO Hack. (2016). http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community *(Onward! 2014)*. ACM, New York, NY, USA, 283–299. https://doi.org/10.1145/2661136.2661156

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 *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11)*. ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2089155.2089159

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)*.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 713–732.

Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. https://doi.org/10.5210/fm.v2i9.548

The Linux Foundation. 2018. Hyperledger Fabric. (2018). https://www.hyperledger.org/projects/fabric

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 447–458. https://doi.org/10.1145/1926385.1926436

Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. 347–359.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2017. Design and implementation of Concurrent C0. *arXiv preprint arXiv:1701.04929* (2017).

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)*.