OBSIDIAN: A LANGUAGE FOR SMART CONTRACTS DESIGNED FOR SAFETY AND USABILITY

Michael Coblenz, Jonathan Aldrich, Brad Myers, Joshua Sunshine
SAFER SMART CONTRACTS

Smart Contracts
• Automate transactions
• Mutually untrusting parties
• Tamper-resistant
• Often buggy
  • DAO reentrancy bug: $50 million stolen
  • Other examples involving lost assets, etc.

The Obsidian Language
• A safer contract language
• Typestate
  • Mitigates misuse bugs, such as in the DAO
• Linearity
  • Mitigates lost asset bugs
• Research goals
  • Mathematical safety
  • Developer usability
main asset contract TinyVendingMachine {
    Coins@Owned coinBin;

    state Full {
        Candy@Owned inventory;
    }

    state Empty;

    transaction buy( TinyVendingMachine @ Full >> Empty this,
                     Coin @ Owned >> Unowned c)
        returns Candy@Owned {
            coinBin.deposit(c);
            Candy result = inventory;
            -> Empty;
            return result;
        }

    ...
main contract TinyVendingMachineClient {
    transaction main(remote TinyVendingMachine@Shared machine) {
        restock(machine);
        if (machine in Full) {
            Coin c = new Coin();
            remote Candy candy = machine.buy(c);
            eat(candy);
        }
    }

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

    private transaction eat(remote Candy @ Owned >> Unowned c) {
        disown c;
    }
}
TECHNICAL CHALLENGE: TYPESTATE AND ALIASING

Offered

Account

Bond @ Offered b

Bond

Account

Bond @ Offered b
TECHNICAL CHALLENGE: TYPESTATE AND ALIASING

If there is a typestate-guaranteeing reference, then all other references must not change typestate.
INITIAL DESIGN
(BASED ON PRIOR WORK)

- Example Annotation: **Owned** Bond@**Sold**

  Permission  Typestate

- Benefit: orthogonal permission and typestate specifications

- Permission semantics
ORTHOGONAL OWNERSHIP AND TYPESTATE

• Tried the initial design with users
  • 6 student participants in lab study
  • Asked participants to fix a typestate- and ownership-related bug
    • Allowing duplicate prescription refills due to ownership problem
• Result: **users had serious difficulties**
  • Viewed types as **complex**: “I haven’t seen...types that complex in an actual language...enforced at compile time.”
  • Had **trouble accurately simulating the compiler**
  • Ownership in particular was **confusing**
    • Participants thought about it dynamically rather than statically
    • Expanded tutorial and practice did not seem to help
• What can we conclude from this?
  • Need design to be a **simple** as possible
  • Minimize reasoning about ownership (separate from typestate)
REVISED DESIGN COMBINES TYPESTATE AND OWNERSHIP

- Owned is used for assets without typestate

- Naming a typestate implies ownership semantically

Example Annotations:
- Money@ Owned
- Switch@ On

Simplifies design
Avoids reasoning about ownership separate from typestate
CASE STUDY: PARAMETRIC INSURANCE

Real client: the World Bank is interested in blockchain-based parametric insurance for use in countries where trust in institutions is weak.
Obsidian is shorter overall

Cost, expiration in every state; must be repeatedly reinitialized

Explicit, annoying state checks (easy to omit)

Manual state tracking
SUMMATIVE USER STUDY

- Goal: evaluate usability
  - Can developers use Obsidian to write real contracts?
- 6 users, 3 hours, Obsidian tutorial + 3 tasks
  - All completed the tutorial and first task successfully
  - Compiler caught several bugs; most were fixed correctly
  - 4 completed second task, 2 completed third task
    - Others ran out of time, but were making progress
  - Subjects who finished scored well on Java pre-test
- Hypothesis: users with good knowledge of OOP can learn and use Obsidian effectively after 90 minute tutorial
- Next: comparison between Obsidian and Solidity
OBSIDIAN DESIGN: SAFETY AND USABILITY

• An **interdisciplinary** approach to smart contract language design
  
  • Smart contract domain drives the high-level design, properties
  
  • Iterative qualitative studies identify usability issues
  
  • PL theory verifies desired properties hold
  
  • Case studies, summative user studies demonstrate expressiveness, usability