Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, Joshua Sunshine

PLs Are User Interfaces

- A PL is a *user interface* for programmers to accomplish their goals
- Therefore, PLs should be amenable to HCI techniques!
- Today, I will show how we used HCI techniques to **design** and **evaluate** a new PL.
- Goal: help ordinary programmers obtain strong safety guarantees
- Bottom line:
  - Sophisticated type systems can *both* guarantee soundness *and* be usable.
  - Methods we developed were useful for iterating on and evaluating the language.

Blockchains and Smart Contracts

Blockchain

- Distributed ledger
- For parties that have not established trust

Smart Contracts

- Programs that process transactions against blockchain state
- Examples
  - Bonds, insurance
  - Gambling
  - Supply chain

Smart Contract Security

- The DAO bug: $50 million stolen + hard fork
- Parity bug: $30 million stolen + frantic workaround
- "…Fourth, some blame for this bug lies with the Solidity language…" [1]
- Programming is hard. How can languages prevent bugs?

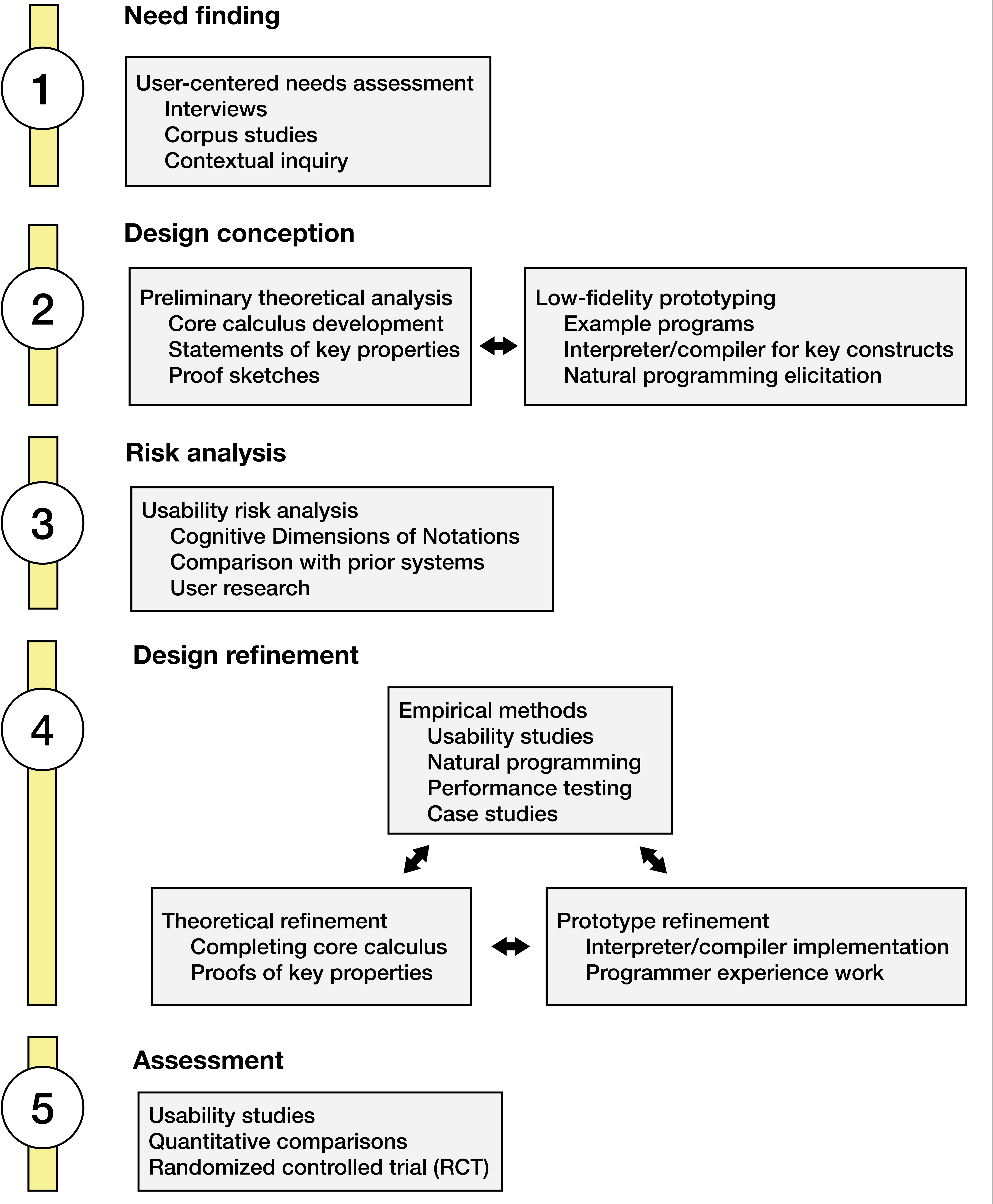
[1] https://paritytech.io/the-multi-sig-hack-a-postmortem/

Obsidian

Overhauling **B**lockchains with **S**tates to **I**mprove **D**evelopment of **I**nteractive **A**pplication **N**otation

https://en.wikipedia.org/wiki/File:Lipari-Obsidienne_(5).jpg

PLIERS: **P**rogramming **L**anguage **I**terative **E**valuation and **R**efinement **S**ystem



Design Ideas

- Blockchain applications frequently:
  - Support different operations depending on *state*
    - Note: DAO hack resulted, in part, from unexpected, reentrant operations [DAO 2016]

Typestate

[DeLine 2004]

- Manage important ***assets***, such as virtual currencies
  - Some smart contract bugs have involved trapped/ forgotten assets [Delmolino et al. 2015]

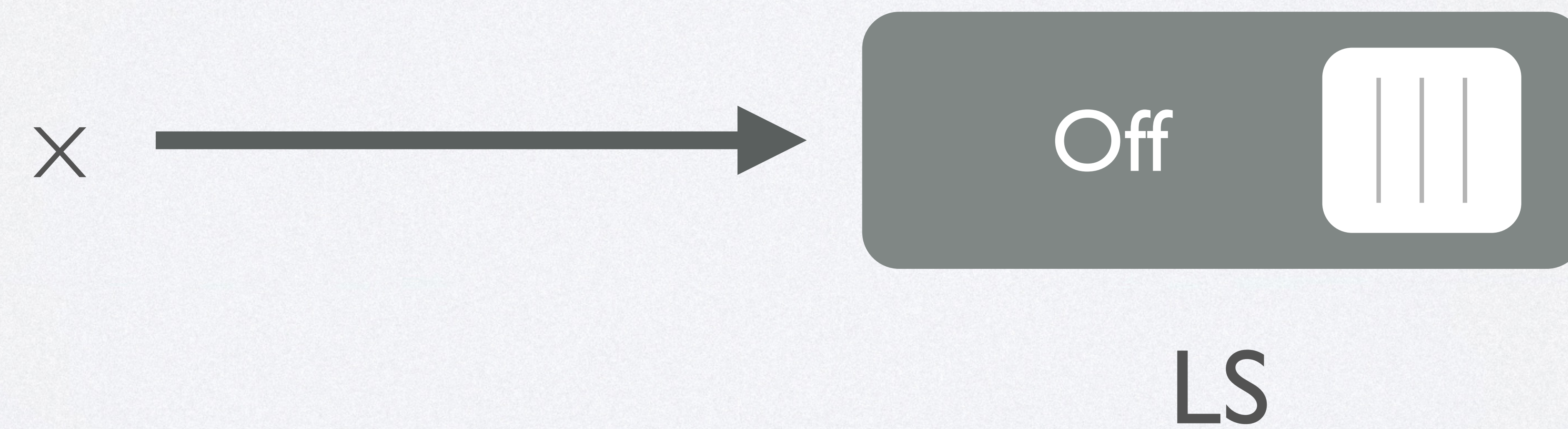
Linearity

[Wadler 1990, Girard 1987]

- This combination is new, and neither technique had been shown to be usable

Without Typestate

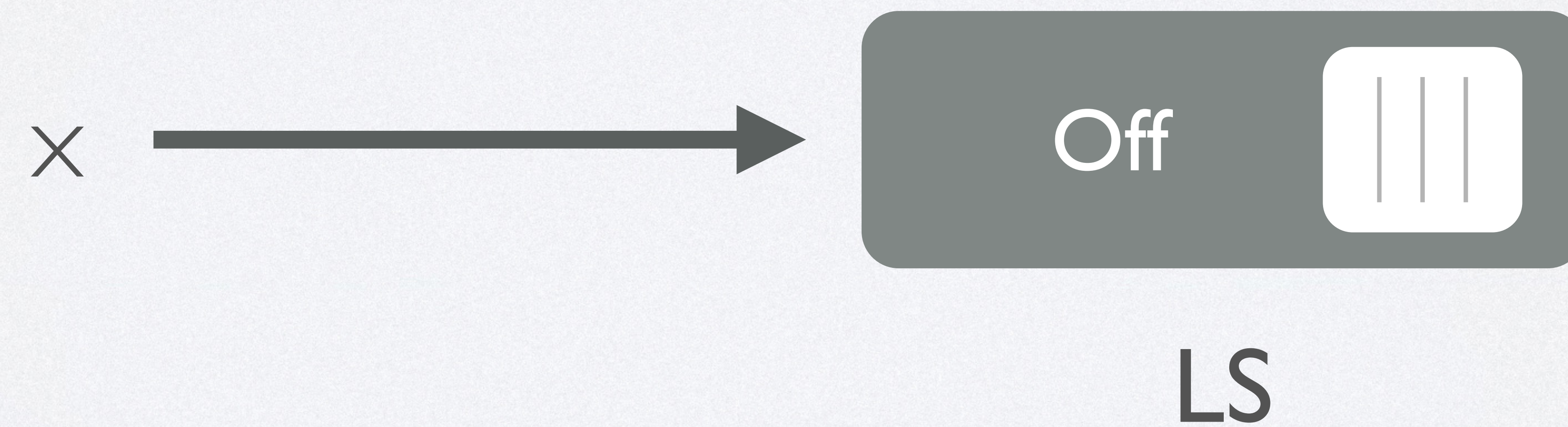
- Type *lacks* state information
- `LightSwitch x = …`



With Typestate

- Type *includes* state information

- `LightSwitch@Off x = …`



Without Linearity

```
Money m = …

transferMoney(m, alice);

transferMoney(m, bob);
```

Compiler says OK!

With Linearity

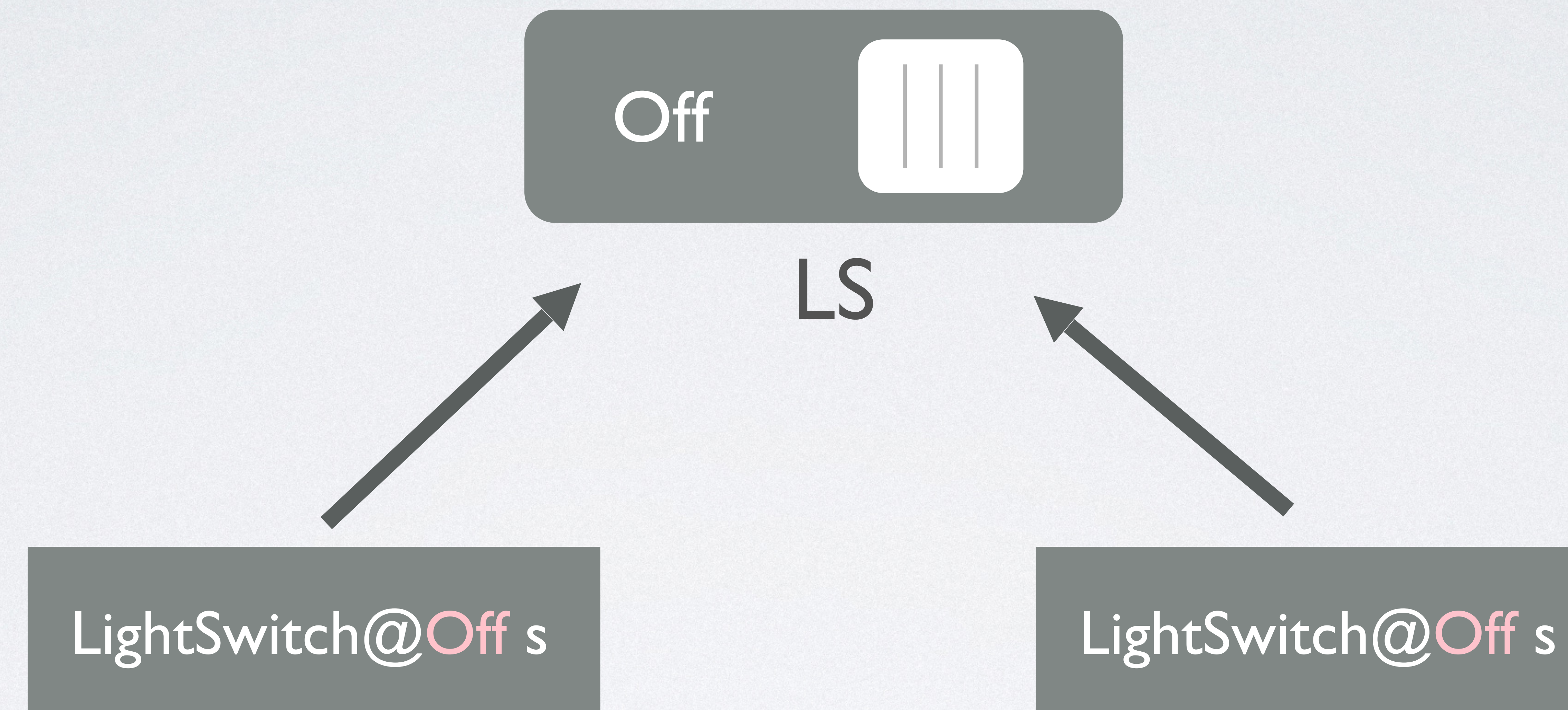
```
owned Money m = …

transferMoney(m, alice);

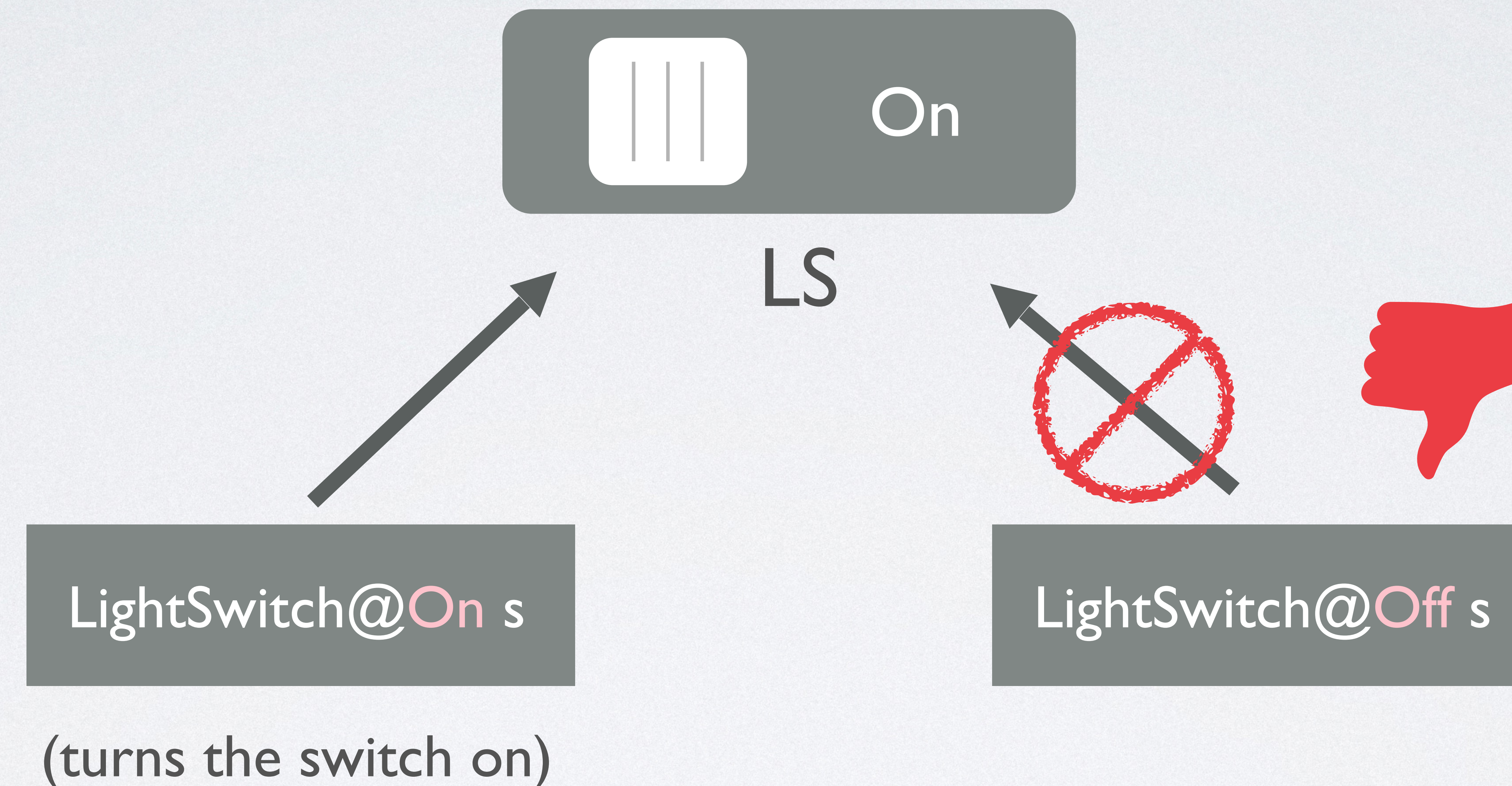
transferMoney(m, bob);
```

Compiler says ERROR!

Technical Challenge: Typestate and Aliasing



Technical Challenge: Typestate and Aliasing



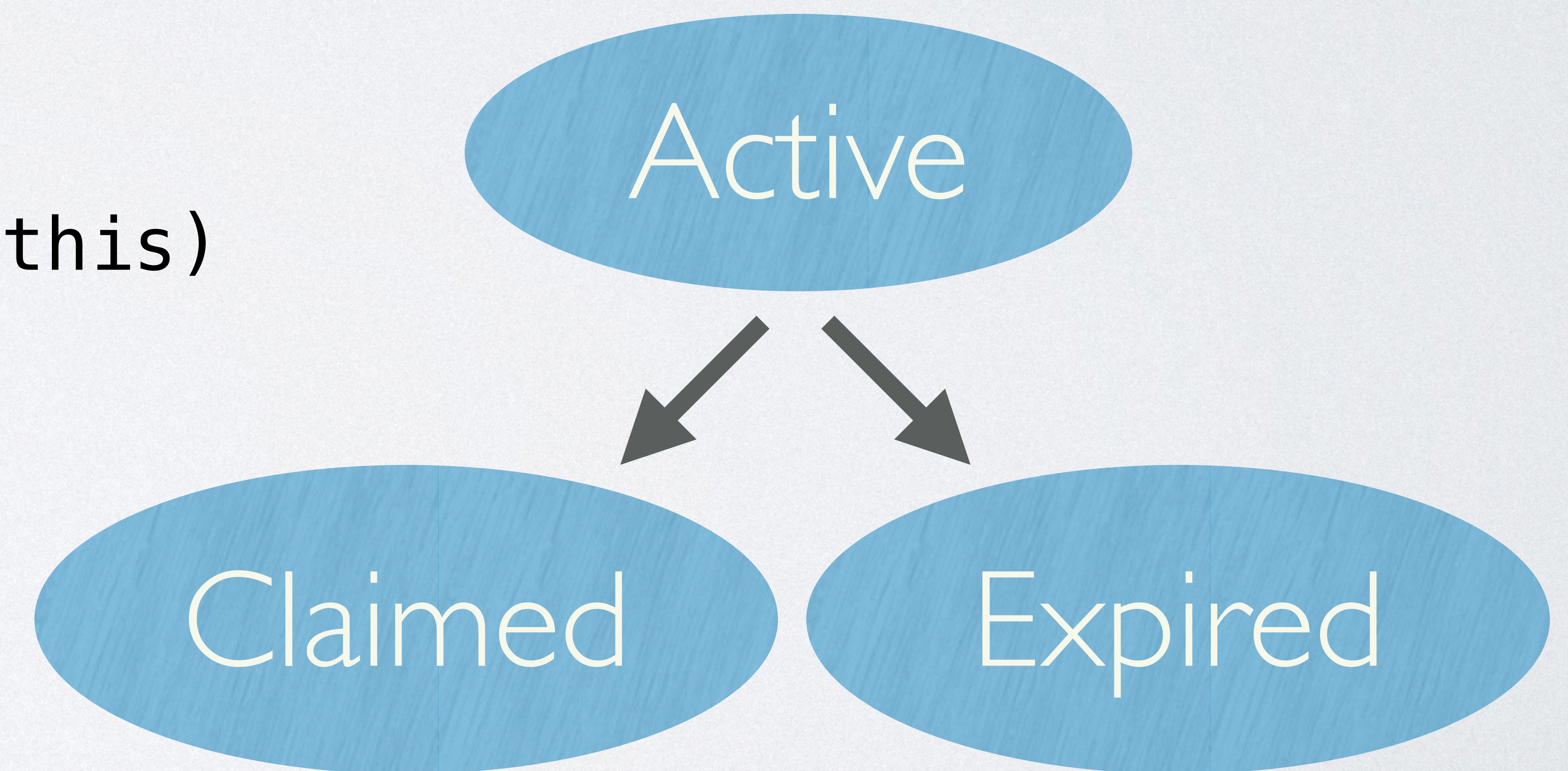
If there is a typestate-specifying reference, then all other references must not change typestate.

Obsidian Example

```
contract InsurancePolicy {
    state Active {
        Money @ Owned benefit;
    }
    state Claimed;
    state Expired;

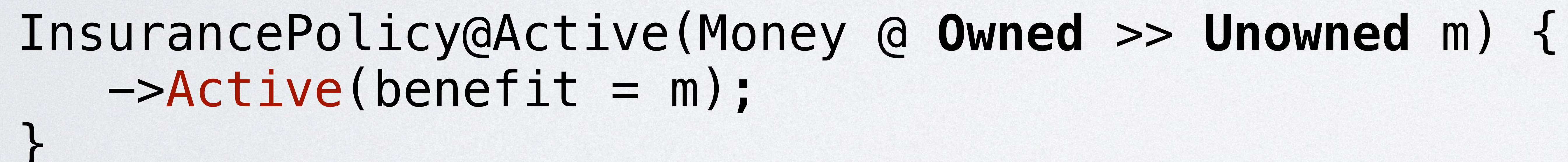
    InsurancePolicy@Active(Money @ Owned >> Unowned m) {
        ->Active(benefit = m);
    }

    transaction claim(InsurancePolicy @ Active >> Claimed this)
                      returns Money @ Owned
    {
        Money result = benefit;
        ->Claimed;
        return result;
    }
}
```



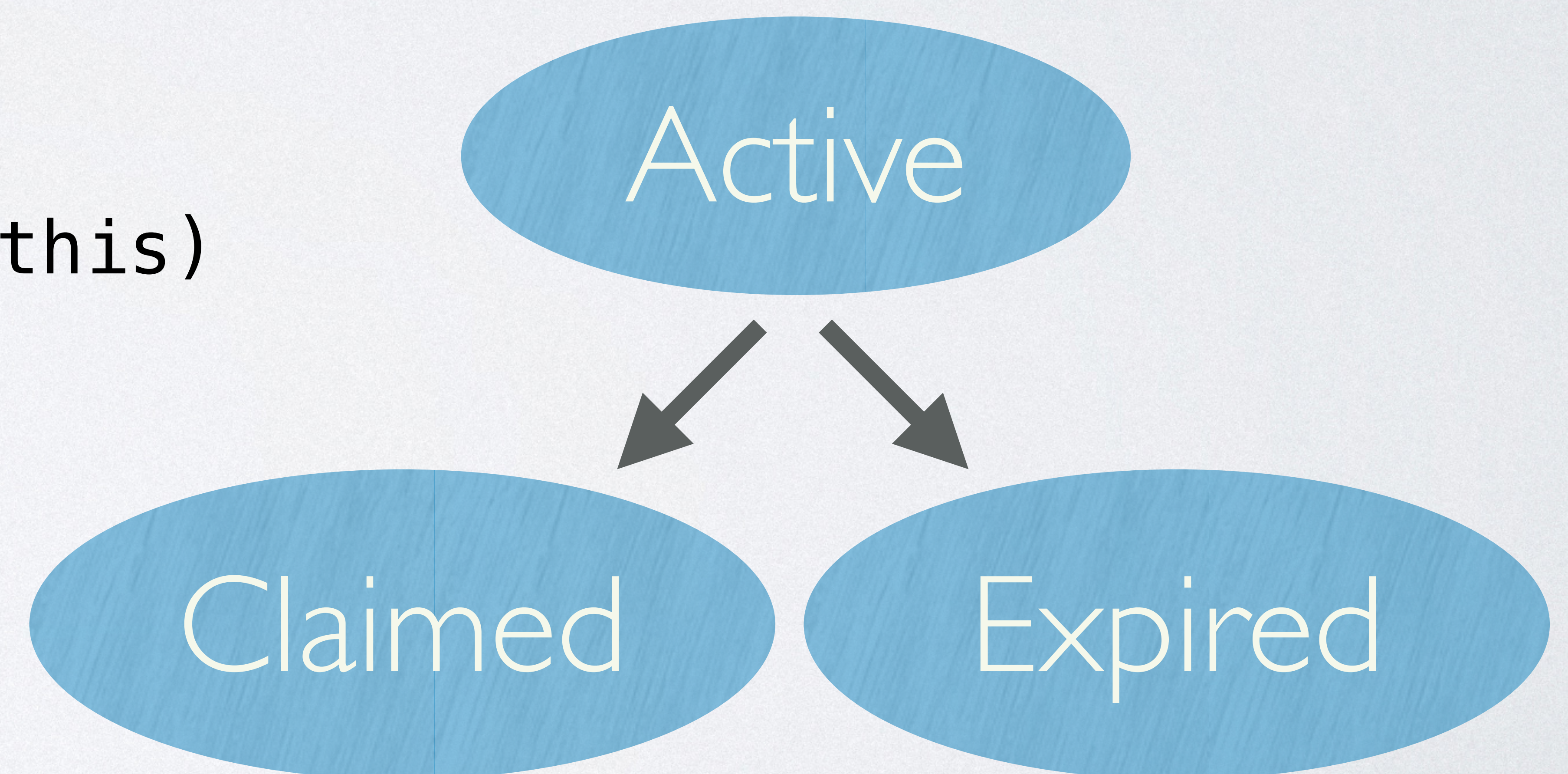
Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;
```



```
   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                    returns Money @ Owned
   {
      Money result = benefit;
      ->Claimed;
      return result;
   }
}
```

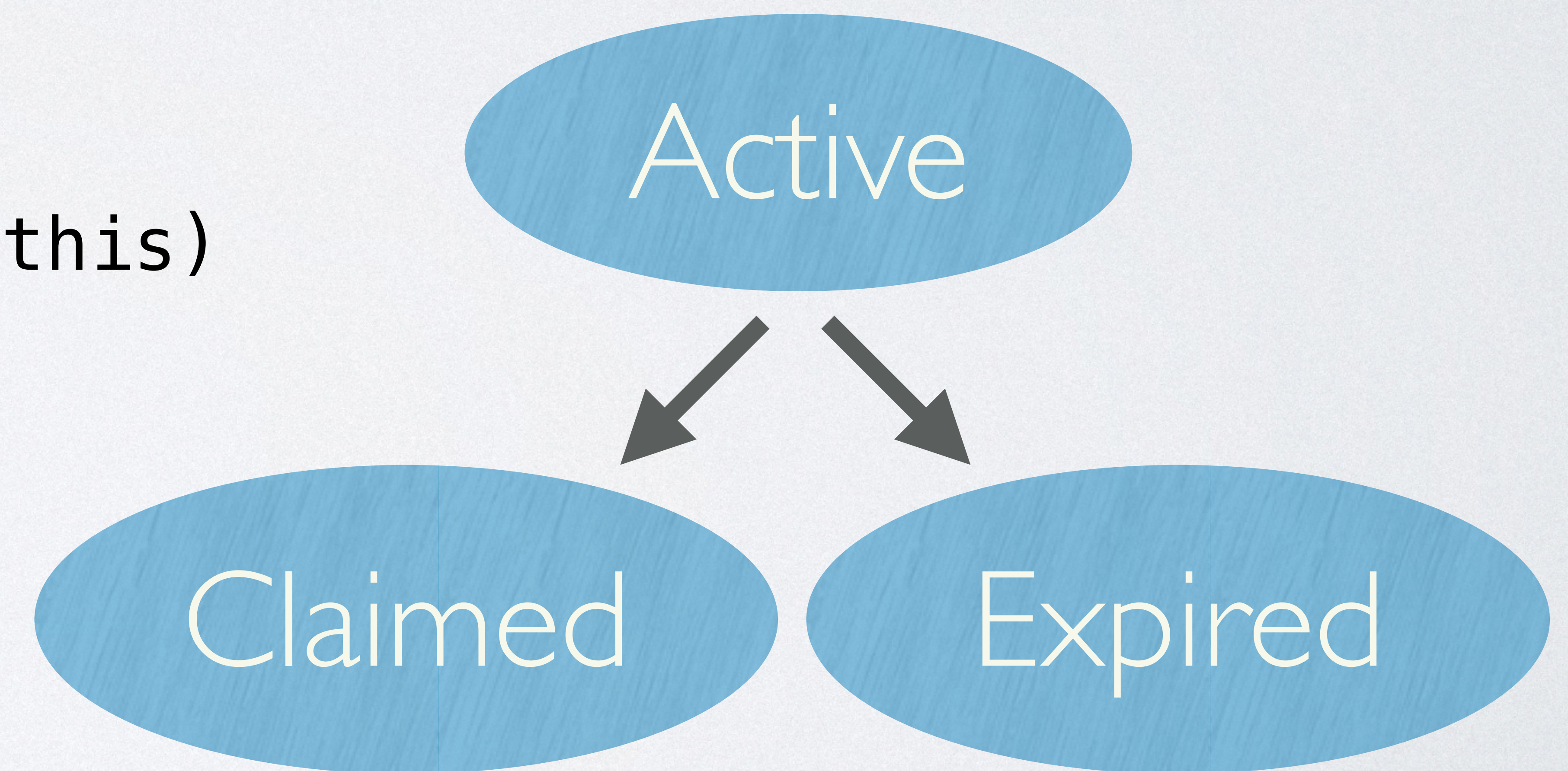


Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;

   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                 returns Money @ Owned
   {
      Money result = benefit;
      ->Claimed;
      return result;
   }
}
```

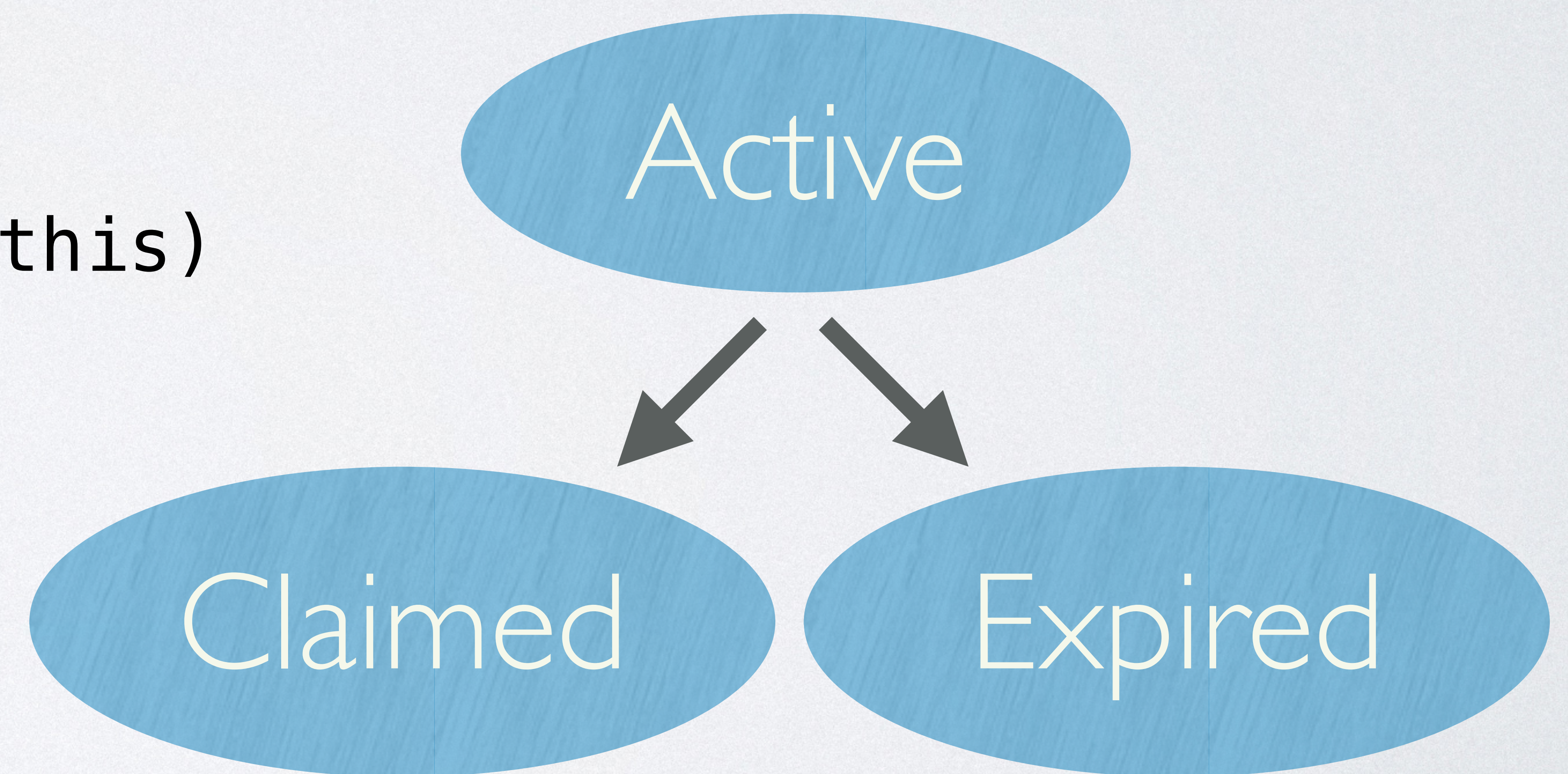


Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;

   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                     returns Money @ Owned
   {
      Money result = benefit;
      ->Claimed;
      return result;
   }
}
```

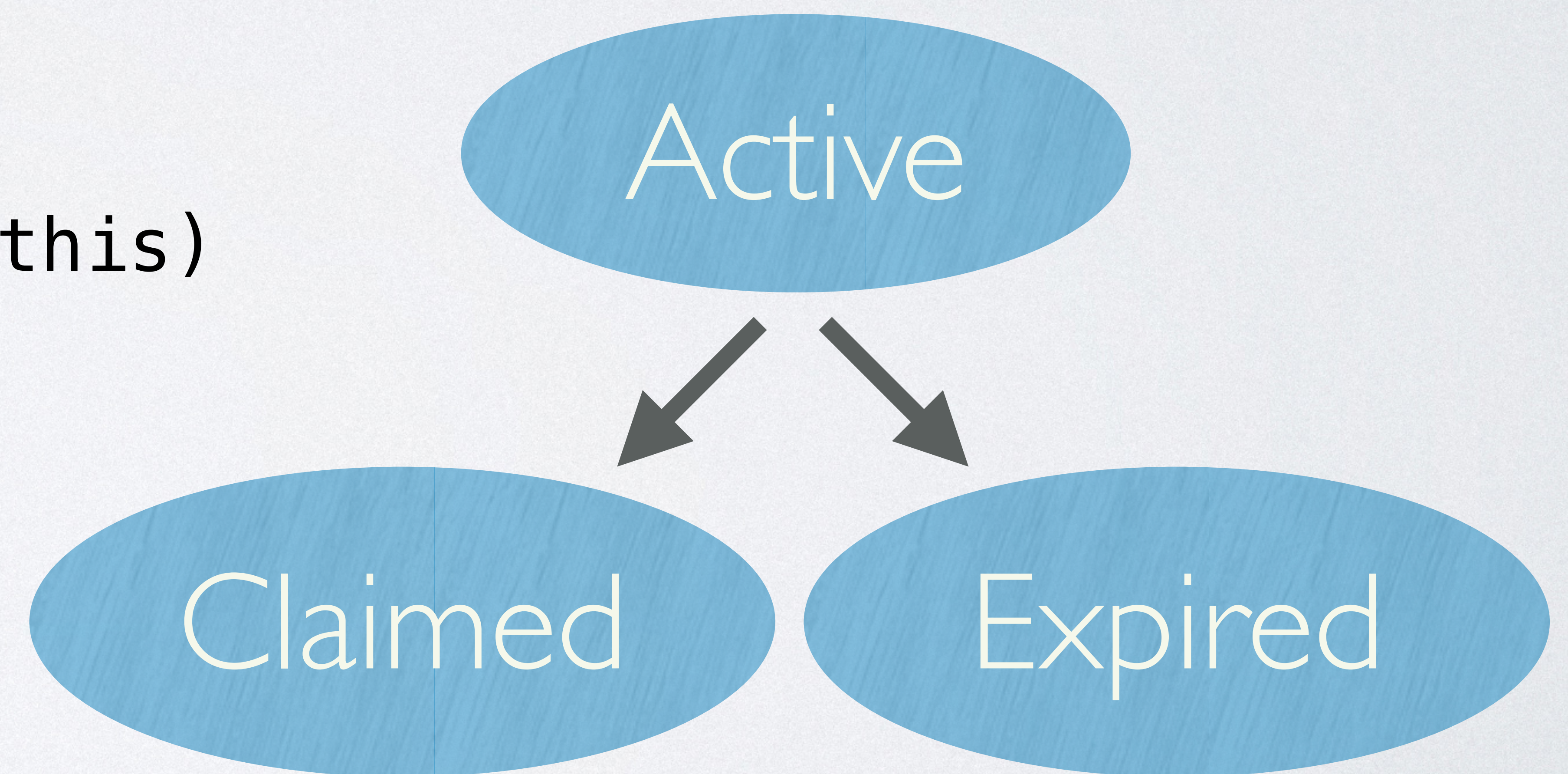


Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;

   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                     returns Money @ Owned
   {
      Money result = benefit;
      ->Claimed;
      return result;
   }
}
```

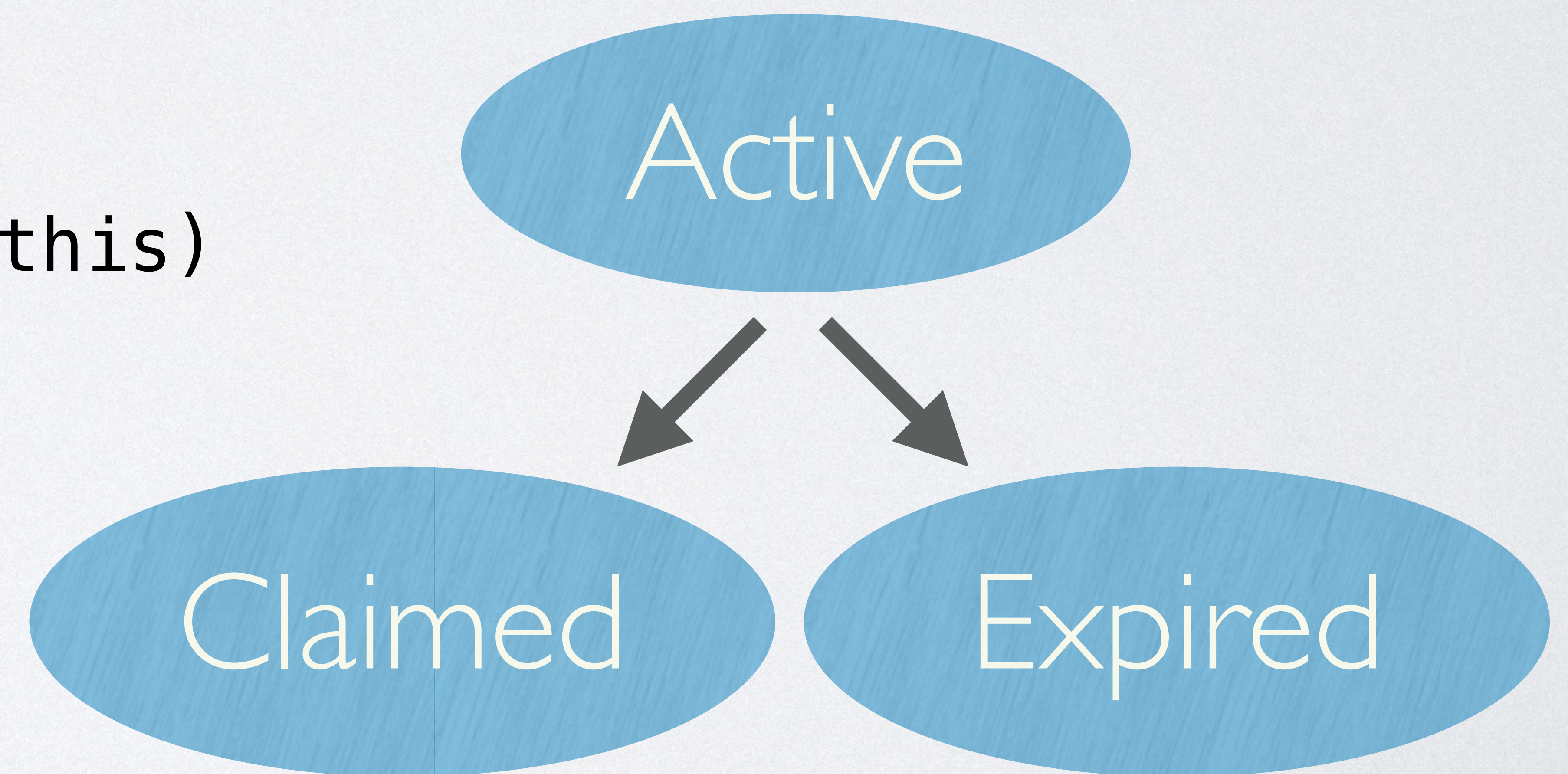


Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;

   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                 returns Money @ Owned
   {
      Money result = benefit;
☞     ->Claimed;
      return result;
   }
}
```

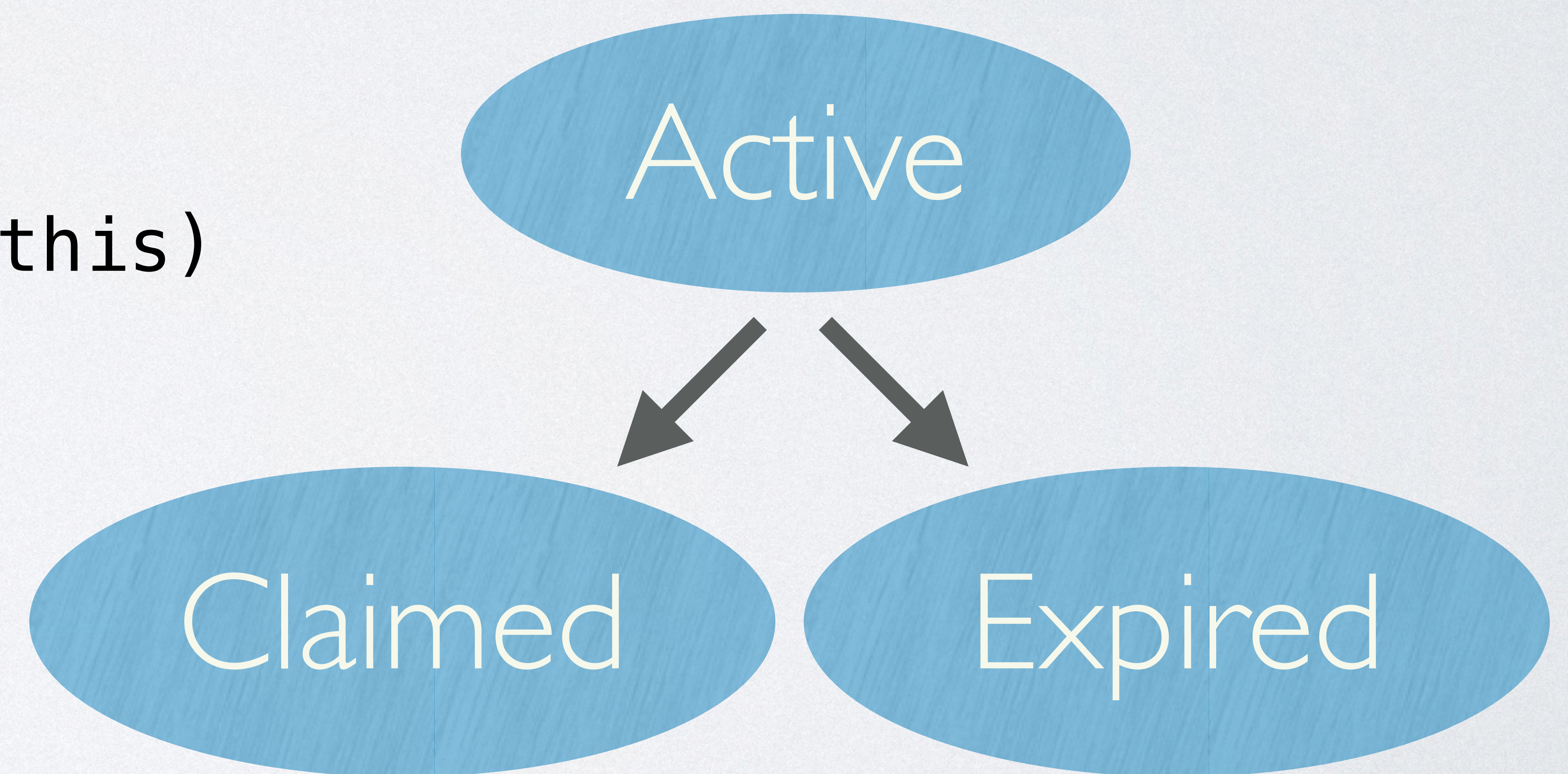


Obsidian Example

```
contract InsurancePolicy {
   state Active {
      Money @ Owned benefit;
   }
   state Claimed;
   state Expired;

   InsurancePolicy@Active(Money @ Owned >> Unowned m) {
      ->Active(benefit = m);
   }

   transaction claim(InsurancePolicy @ Active >> Claimed this)
                     returns Money @ Owned
   {
      Money result = benefit;
      ->Claimed;
☞     return result;
   }
}
```



Quantitative Study

- Is Obsidian *better* than Solidity:
  - First, can we conduct a user study in an unfamiliar language at all?
    - We'd have to recruit and train participants…
  - Are people able to complete tasks in Obsidian despite the complex type system?
  - Do Solidity users insert the kinds of bugs that Obsidian detects?

Participants

- N=20 participants (14 M, 6 F)
- Medians:
  - 6 years programming experience (9 Solidity, 5 Obsidian)
  - 1 year professional experience (1 Solidity, 1 Obsidian)
  - 2 years Java experience (2 Solidity, 1.5 Obsidian)

Procedure

- Tutorial on their assigned condition
  - with practice problems and compiler
  - questions answered
- Three programming tasks
  - no questions allowed
  - compiler only — no runtime enviornment
- Four hours. Paid with $75 Amazon gift certificate.

Tutorial

Obsidian
latest

Search docs

- Getting Started
- Obsidian Language Tutorial
  - Ownership – Introduction
    - Principles of ownership
  - Ownership – Transactions
  - Ownership – Variables
  - Ownership – Miscellaneous
  - Assets
  - States – Introduction
  - States – Manipulating State
  - States – Miscellaneous
  - States and Assets
  - Using Obsidian on a Blockchain
  - Taking Advantage of Ownership
- Obsidian Reference

Docs » Obsidian Tutorial » Ownership – Introduction

Edit on GitHub

Ownership – Introduction

Principles of ownership

Our new programming language is object-oriented. It includes *contracts*, which are like classes, and can have *fields* and *transactions*, analogous to Java fields and methods respectively. An Obsidian program must have exactly one `main contract`. In addition, of the many variables or fields that reference objects, exactly *one* of them can own the object, as seen in diagram *(a)* below. An object can have any number of Unowned references, and, if the object is not Owned, it can have any number of Shared references (shown in *(b)* below). An object with Shared references can also have Unowned references, but not Owned ones.

	Solidity	Obsidian
Avg. time	86 mins	98 mins
Standard deviation	28 mins	21 mins

Task Objectives

- Reflect use cases of community interest
- Range of difficulties
- Assess: do Solidity participants lose assets? Can Obsidian participants get work done?
- Assess: could Obsidian participants successfully use ownership for security? If so, is it faster than using dynamic enforcement?
- Assess: how do Solidity and Obsidian compare in an open-ended programming task?

Condition: Solidity
Time limit: 30 minutes

Auction Task

```
contract Auction {
    address maxBidder; // the bidder who made the highest bid so far
    uint maxBidAmount;
    // 'payable' indicates that we can transfer money to this address
    address payable seller;

    // Allow withdrawing previous bid money for bids that were outbid
    mapping(address => uint) pendingReturns;

    enum State { Open, BidsMade, Closed }
    State state;

    constructor(address payable s) public {
        seller = s;
        state = State.Open;
    }
```

implements *withdrawal pattern*

RQ: Do Solidity participants lose assets? Can Obsidian participants achieve goals?

```
function bid() public payable {
        if (state == State.Open) {
            maxBidder = msg.sender;
            maxBidAmount = msg.value;
            state = State.BidsMade;
        }
        else {
            if (state == State.BidsMade) {
                //if the newBid is > than the current Bid
                if (msg.value > maxBidAmount) {
                    //1. TODO: fill this in.


                }
                else {
                    //2. TODO: return the newBid money to the bidder,
                    //   since the newBid wasn't high enough.
                    //   You may call any other functions as needed.

               }
            }
            else {
                revert ("Can only make a bid on an open auction.");
            }
        }
    }
```

```
function bid() public payable {
        if (state == State.Open) {
            maxBidder = msg.sender;
            maxBidAmount = msg.value;
            state = State.BidsMade;
        }
        else {
            if (state == State.BidsMade) {
                //if the newBid is > than the current Bid
                if (msg.value > maxBidAmount) {
                    //1. TODO: fill this in.
                    maxBidder = msg.sender;
                    maxBidAmount = msg.value;
                }
                else {
                    //2. TODO: return the newBid money to the bidder,
                    //   since the newBid wasn't high enough.
                    //   You may call any other functions as needed.
                    msg.sender.transfer(msg.value);
                }
            }
            else {
                revert ("Can only make a bid on an open auction.");
            }
        }
    }
```

Forgot refund!

OK, but neglects withdrawal pattern

(P48)

Results: Successes

	Solidity (N=10)	Obsidian (N=10)
Finished in 30 mins	9	8
Correct answer	2	6

2 of these corrected lost asset errors

($p \approx 0.09$)

```
Variable 'maxBid' is an owning reference to an
asset, so it cannot be overwritten.
```

Results: Failures (Among Completions)

```
pendingReturns[maxBidder] = maxBidAmount;
```

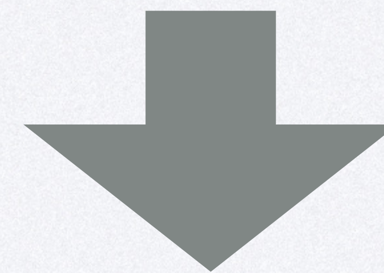
	Solidity (N=10)	Obsidian (N=10)
Overwrote prior refund	4	0
Forgot refund	3	0

Prescription Task

- Problem: how to enforce that a `Prescription` is only deposited once in a `Pharmacy`

- Solution: transfer ownership to `Pharmacy`

```
transaction depositPrescription(Prescription@Shared p)
    returns int {…}
```



```
transaction depositPrescription(Prescription@Owned >> Unowned p)
    returns int {…}
```

Prescription in Solidity

- Need global state tracking registrations.

Time Limit

- 35 minutes.

Prescription Results

	Solidity	Obsidian
Correct dynamic solution	2	1
Correct static solution	N/A	6

Threats to Validity

- Lab study
- Four hours
- Students
- Tasks modeled after real-world examples but not necessarily representative

Observations

- Experiments with sophisticated type systems are practical!
  - Features are teachable in a consistent way
  - Participants leveraged features effectively
  - Abuse of `disown` shows opportunities for improvement

Keys to Success

- Iteratively design and pilot documentation and tasks
- Draw tasks from real-world contexts (external validity)
- Recruit appropriately (e.g. we recruited Master's students)
- Tutorial should include *practice* and *assessment*

Obsidian RCT Conclusion

- You, too, can evaluate your language design with randomized controlled trials (RCTs)
- The road is long (about six months for Obsidian — piloting is required!). But:
  - No other way to know your work actually benefits people other than yourself
  - Build and test associated materials (e.g., documentation) along the way
  - Identify opportunities for improvement along the way

BACK-PORTING DESIGN CHOICES

- Wanted to assess usability of typestate
- Teach people Obsidian?
- No, add typestate to Java.
- Can do this without an implementation (Wizard of Oz)!

Orthogonal Ownership and Typestate

```
owned Prescription@Full
```

- N=6 students in lab study (using Java annotations)
  - Asked participants to fix a typestate- and ownership-related bug
  - Allowing duplicate prescription refills
- Result: users had serious difficulties
  - *"I haven't seen...types that complex in an actual language...enforced at compile time."*
  - Participants thought about ownership dynamically rather than statically
  - Expanded tutorial and practice did not seem to help

Assignment and Parameter-Passing

- Initial design: **`@Owned`** means acquires ownership

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

- User study results (N=6): this is confusing
  - "when I [annotate this constructor type **`@Owned`**], I'm not sure if I'm making a variable owned or I'm transferring ownership."
- People expected (non-modular) interprocedural analysis
- Revised design: make change in ownership explicit

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

Type Declarations

- Initial design (from prior work):
  - Always specify typestate when declaring variable

```
LightSwitch@On s = …
```

- User study results (N=5): Too confusing!
  - ```
    LightSwitch@Off s1 = new LightSwitch();
    s1.turnOn();
    ```
- Revised design
  - Specify initial typestate, and any transition, in method signature (modularity)
  - No typestate on local variables; support static typestate assertions

```
[s1 @ Off];
```
```

CONCLUSION

- Usability studies can
  - reveal serious problems, leading to design improvements
  - help you prepare for an RCT
- Teaching people strong type systems well enough to obtain results can be done quickly
- But user studies are expensive — choose RQs wisely!

FAQ (1)

- If a usability study goes great, is the language usable?
  - No, you need an RCT for that.
- You got the wrong participants.
  - At least the results may generalize to the population from which I drew them.

FAQ (2)

- You didn't get enough participants.
  - I will never be able to find *all* the obstacles people will face.
  - If my RCT got significant results, then you should complain about my external validity instead.
- Your changes might make the language better for novices but worse for experts.
  - Maybe, but theory can help evaluate this question.