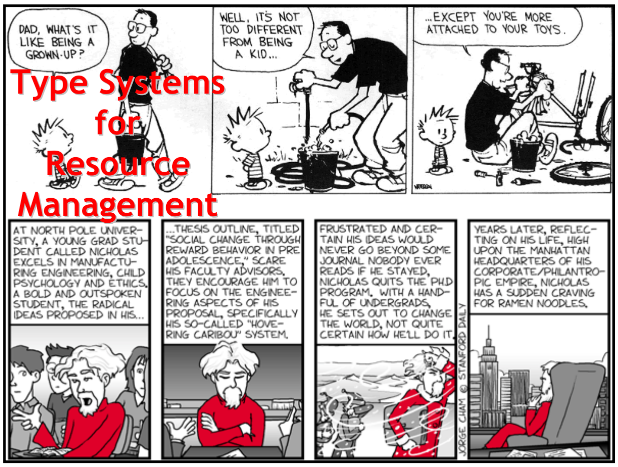
# Type Systems for Resource Management


DAD, WHAT'S IT LIKE BEING A GROWN-UP?
WELL, IT'S NOT TOO DIFFERENT FROM BEING A KID...
...EXCEPT YOU'RE MORE ATTACHED TO YOUR TOYS.
WATTERSON
AT NORTH POLE UNIVERSITY, A YOUNG GRAD STUDENT CALLED NICHOLAS EXCELS IN MANUFACTURING ENGINEERING, CHILD PSYCHOLOGY AND ETHICS. A BOLD AND OUTSPOKEN STUDENT, THE RADICAL IDEAS PROPOSED IN HIS...
...THESIS OUTLINE, TITLED "SOCIAL CHANGE THROUGH REWARD BEHAVIOR IN PRE ADOLESCENCE," SCARE HIS FACULTY ADVISORS. THEY ENCOURAGE HIM TO FOCUS ON THE ENGINEERING ASPECTS OF HIS PROPOSAL, SPECIFICALLY HIS SO-CALLED "HOVERING CARIBOU" SYSTEM.
FRUSTRATED AND CERTAIN HIS IDEAS WOULD NEVER GO BEYOND SOME JOURNAL NOBODY EVER READS IF HE STAYED, NICHOLAS QUITS THE PHD PROGRAM. WITH A HANDFUL OF UNDERGRADS, HE SETS OUT TO CHANGE THE WORLD, NOT QUITE CERTAIN HOW HE'LL DO IT.
JORGE CHAM © STANFORD DAILY
YEARS LATER, REFLECTING ON HIS LIFE, HIGH UPON THE MANHATTAN HEADQUARTERS OF HIS CORPORATE/PHILANTROPIC EMPIRE, NICHOLAS HAS A SUDDEN CRAVING FOR RAMEN NOODLES.

## Take-Home Message

- Type systems can be used to track how programs uses resources and APIs.
- Linear type systems track objects precisely but forbid aliasing. They are like dataflow analyses or operational semantics.
- Programmers are often poor at handling run-time errors correctly.

## Linear Type Systems

- Type systems for managing resources are usually linear
- From linear logic, where each hypothesis must be used (discharged) *exactly once*
- Each important object in a linear type system must be *freed exactly once*
- Each important object is known by a *unique name* so that it can be tracked

## Linear Type System Drawbacks

- Perfect alias resolution is undecideable
- So we can never put an important object in memory in a way that allows it to be aliased
  - Or we might free it 0 or 2 times
- Thus unique names cannot be *p or “the ship’s doctor” but must instead be “local variable mysock” or “Worf”

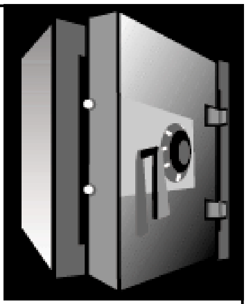
## Typical Linear Type System

- Judgment: $S_1 \vdash \mathrm{cmd} : S_2$
  - S is the current set of resources
  - Linear type systems are flow-sensitive
  - Linear type systems behave like opsem
- Rules:

$$\frac{\{\mathrm{Name}_i\} \in \not{S}_1 \quad S_2 = S_1 \cup \{\mathrm{Name}_i\}}{S_1 \vdash \mathbf{new}\ \mathrm{Name}_i : S_2}$$

$$\frac{S_1 \vdash c_1 : S_2 \qquad S_2 \vdash c_2 : S_3}{S_1 \vdash c_1\ ;\ c_2 : S_3} \qquad \frac{S_2 = S_1 \setminus \{\mathrm{Name}_i\}}{S_1 \vdash \mathbf{del}\ \mathrm{Name}_i : S_2}$$

## Topic: Vault



- There are easily two Vault papers; I will skim.

## Enter the Vault

- Vault is a novel programming language
  - Designed ~2001 by Manuel Fähndrich and Rob DeLine at Microsoft Research
- Vault allows you to describe and statically enforce resource management protocols
- Vault can prevent resource leaks and API violations
- Vault is based on linear type systems
  - In a linear type system, each resource must be used *exactly once*.

## Tracking Individual Objects

Rule 1: “Close every socket that you open.”

Rule 2: “Do not read from a socket after closing it.”

```
void ReadFromSocks (SOCKET s1, SOCKET s2) {
  …
  read(s1,buf,n);
  close(s1);
  read(s2,buf,n);
  close(s2);
}
```

Are we obeying the rules?

## Vault Intution

- *At every program point we will keep track of exactly which sockets you have and whether each one is opened or closed*

- Every point = flow-sensitive analysis
- Sockets = all important resources
- Exactly which = “named objects” or “keys”
- Opened or closed = typestate of that object
- The type system *is* a dataflow analysis!

## Vault Heap Properties

- New notions: tracked and guarded objects
- Single pointer to each tracked object
  - Tracked objects form linear regular trees
  - *No aliasing is possible* with tracked objects
  - Tracked objects have names (names = “keys”)
- Any number of pointers to guarded objects
  - Many guarded objects inside one tracked object
  - Not covered in this talk
- Goals:
  - Model state of tracked objects statically
  - Allow explicit but checked malloc/free

## Interfaces and Tracked Types

- Tracked Type Syntax: tracked(*K*) *T*
  - Can do free and cast (unless type T is abstract)
  - Can do normal operations on type T
  - But *only when* key K is accessible (= we hold K)
- Vault functions change the key set
  - Change spec = function pre- and post conditions
  - Add key `tracked(K) T foo() [new K];`
  - Remove key `void bar(tracked(K) T) [-K];`
  - Change state `void baz(tracked(K) T) [K(T)->(S)];`
- Language primitives also change the key set
  - Allocation expression adds a key `new tracked T`
  - Deallocation expression removes a key `free e`

## Sockets in Vault

```
tracked(S) sock socket(domain, comm_style,
int)[ new S @ raw ];
```

```
void bind(tracked(S) sock, sockaddr)
[ S @ raw -> named ];
```

```
void listen(tracked(S) sock, int)
[ S @ named -> listening ];
```

```
tracked(N) sock accept(tracked(S) sock,
sockaddr)
[ S @ listening, new N @ ready ];
```

```
void receive(tracked(S) sock, byte[])
[ S @ ready ];
```

```
void close(tracked(S) sock) [ -S @ _ ];
```

## Socket Client

```
• void work() {                        { }
    tracked sock s = socket(…);        { S @ raw }
    bind(s, …);                        { S @ named }
    listen(s, …);                      { S @ listening }
    while(true) {                      { S @ listening }
      tracked sock t = accept(s);      { S @ listening, N @ ready}
      receive(t, buf);                 { S @ listening, N @ ready}
      close(t);                        { S @ listening }
    }                                  { S @ listening }
    close(s);                          { }
  }
```

## Vault Typechecking

- A function's key transformer annotation gives its pre- and post-conditions
- On each path through a function, check
  1. Pre-condition is transformed into post-condition
  2. All proof obligations are satisfied
     - Pre-conditions of other function calls
     - Primitive operations (memory access, free)
- Avoid exponential blow-up (state explosion) by
  - requiring *uniform predicate at join points*
  - allowing only simple function specs

## Not In My BackVault

```
void work() {                              { }
  if (p)                                   { }
    tracked sock s = socket(…);            { S @ raw }
  else
    skip;                                  { }
  printf("hello world\n");                 ERROR
}

void aie() {
  DoublyLinkedList * L = NULL;             { }
  while (rand() % 100 > 50) {              { }
    tracked sock s = socket(…);            { S @ raw }
    L = PrependNode(s,L);                  ERROR
  }
}
```

All paths entering a join-point must have the same tracked set.

We could alias s using L.

## Vault Evaluation

- Used for windows device drivers, directx d3d programs, parser combinators …
- More complicated (non-linear) data structures can be handled using the techniques in Paper #2 (adoption and focus)
- Concurrency is difficult (convoluted locking)
- Annotation burden can be high
- "Great design, hard to use"

## Topic:
## Language Support For Managing Resources In Exceptional Situations

- There are easily two WN papers; I will skim.

## Overarching Intuition

- Use simple policies
  - open/close, as in SLAM or Vault
- Use a linear type system for sets of resources
  - Not for individual resources themselves!
- Close all resources along all paths
  - Even those for unexpected exceptions

1. Motivate The Problem
2. Propose A New Language Feature

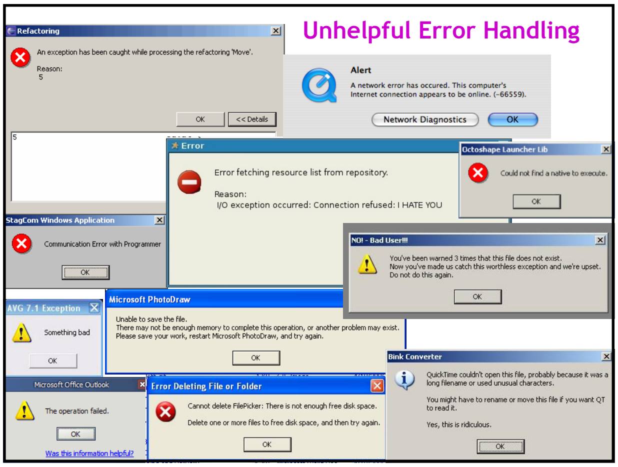
## Defining Terms

- Exceptional Situations:
  - Network problems, DB access errors, OS resource exhaustion, …
- Typical Exception-Handling:
  - Resend packet, show dialog box to user, …
  - Application-specific, don’t care in this lecture
- Exception-Handling Mistakes (Bugs!):
  - One example: a network error occurs and the program forgets to release a database lock with an external shared database, …

## The State Of The Art

- Most Common Exception Handlers
  - #1. Do Nothing
  - #2. Print Stack Trace, Abort Program
- Higher-level invariants should be restored, interface requirements should be respected
  - Aside from handling the exceptional situation, code should clean up after itself
  - What do we mean by “should”?

## Unhelpful Error Handling



**Refactoring**

An exception has been caught while processing the refactoring 'Move'.

Reason:
5

5

**Alert**

A network error has occured. This computer's Internet connection appears to be online. (-66559).

**Error**

Error fetching resource list from repository.

Reason:
I/O exception occurred: Connection refused: I HATE YOU

**Octoshape Launcher Lib**

Could not find a native to execute.

**StagCom Windows Application**

Communication Error with Programmer

**NO! - Bad User!!!**

You've been warned 3 times that this file does not exist.
Now you've made us catch this worthless exception and we're upset.
Do not do this again.

**AVG 7.1 Exception**

Something bad

**Microsoft PhotoDraw**

Unable to save the file.
There may not be enough memory to complete this operation, or another problem may exist.
Please save your work, restart Microsoft PhotoDraw, and try again.

**Bink Converter**

QuickTime couldn't open this file, probably because it was a long filename or used unusual characters.

You might have to rename or move this file if you want QT to read it.

Yes, this is ridiculous.

**Microsoft Office Outlook**

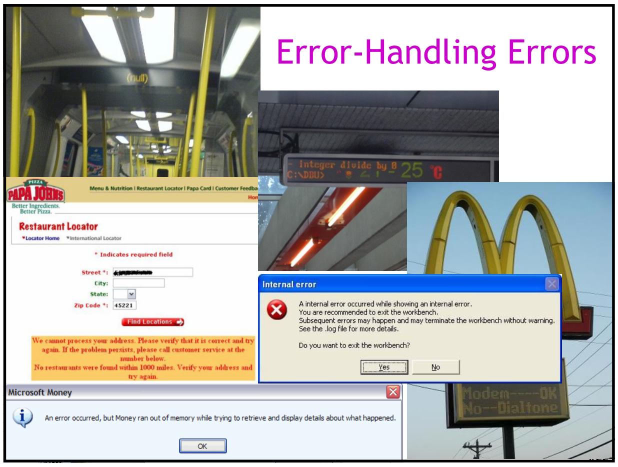
The operation failed.

**Error Deleting File or Folder**

Cannot delete FilePicker: There is not enough free disk space.

Delete one or more files to free disk space, and then try again.

## Error-Handling Errors



* Indicates required field

Street *: [redacted]
City:
State:
Zip Code *: 45221

We cannot process your address. Please verify that it is correct and try again. If the problem persists, please call customer service at the number below.
No restaurants were found within 1000 miles. Verify your address and try again.

**Internal error**

A internal error occurred while showing an internal error.
You are recommended to exit the workbench.
Subsequent errors may happen and may terminate the workbench without warning.
See the .log file for more details.

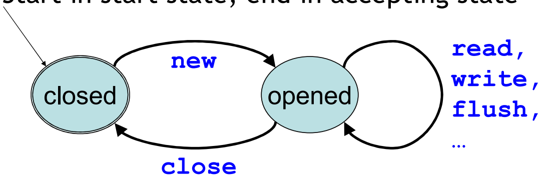
Do you want to exit the workbench?

**Microsoft Money**

An error occurred, but Money ran out of memory while trying to retrieve and display details about what happened.

## Example Safety Policy

- Safety Policy Governing Java Streams
- One FSM per Stream object
- Edges = events in program
- Start in start state, end in accepting state

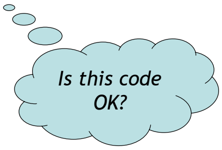


## What’s Up In Real Life?

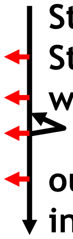
- Now knowing what we should be doing
- It is difficult for programmers to consider all of the possible execution paths in the presence of exceptions
- So there are often a few paths related to exceptional conditions in which the safety policy is violated
- Let’s see an example:

## Simplified Java Code

```
Stream input = new Stream();
Stream output = new Stream();
while (data = input.read())
     output.write(data);
output.close();
input.close();
```

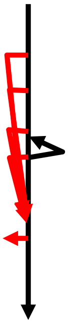


## Error Paths - Hazards



```
Stream input = new Stream();
Stream output = new Stream();
while (data = input.read())
     output.write(data);
output.close();
input.close();
```

## Fix It (?) With Try-Finally



```
try {
  Stream input = new Stream();
  Stream output = new Stream();
  while (data = input.read())
      output.write(data);
} finally {
  output.close();
  input.close();
}
```

## Fix It With Runtime Checks

```
input = output = null;
try {
   input = new Stream();
   output = new Stream();
   while (data = input.read())
     output.write(data);
} finally {
   if (output != null)
     try { output.close(); } catch (Exception e) { }
   if (input != null)
     try { input.close(); } catch (Exception e) { }
}
```

Note: the blue exception-handling code is a big chunk of the program!

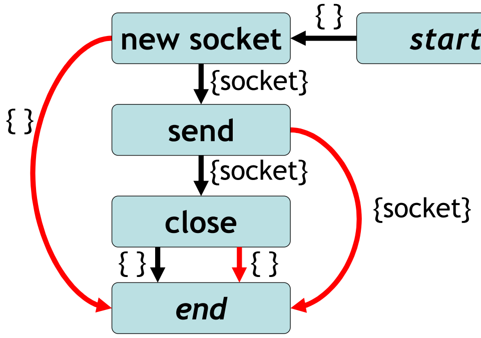
## Finding Exception Handling Bugs

- We consider four generic resources
  - Sockets, files, streams, database locks
  - From a survey of Java code
  - Usually simple two-state safety policies
- Program should release them along all paths, even in exceptional situations
- Exceptional situations are rare …
- So use a static analysis to find mistakes

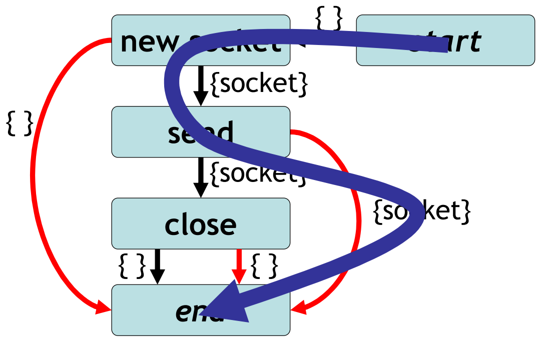
## Analysis Example Program

```
try {
  Socket s = new Socket();
  s.send(“GET index.html”);
  s.close();
} finally { } // bug!
```

## Analysis Example



## Analysis Example



## Analysis Results

| Program Name | Lines of Code | Methods with Exception-Handling Mistakes | Forgotten Resources |
|---|---|---|---|
| jboss | 107k | 40 | DB, Strm |
| mckoi-sql | 118k | 37 | Strm, DB |
| portal | 162k | 39 | DB, File |
| pcgen | 178k | 17 | File |
| compiere | 230k | 322 | DB |
| org.aspectj | 319k | 27 | File, Strm |
| ptolemy2 | 362k | 27 | File, Strm |
| eclipse | 1.6M | 126 | Strm, File |
| … 18 others … | 783k | 183 | File, DB |
| Total | 3.9M | 818 | File, DB |

## Destructors and Finalizers

- Destructors – great for stack-allocated objects
  - But error-handling contains arbitrary code
  - e.g., 17 unique cleanups in undo (34 lines)
- Finalizers – widely reviled
  - Called by GC: too late!
  - No ordering guarantees
  - Programs do not use them (13 user-defined ones in 4M LOC, libraries inconsistent)

## Fix It With Flags

```
int f = 0;                    // flag tracks progress
try {
  openX(); f = 1; work();
  openY(); f = 2; work();
  openZ(); f = 3; work();
} finally {
  switch (f) {                 // note fall-through!
    case 3: try { closeZ(); } catch (Exception e) {}
    case 2: try { closeY(); } catch (Exception e) {}
    case 1: try { closeX(); } catch (Exception e) {}
  }
}
```

## Fix It With Flags (Ouch!)

```
int f = 0;                    // flag tracks progress
try {
  openX(); f = 1; work();
  if (…) { didY = true; openY(); f = 2; } work();
  openZ(); f = 3; work();
} finally {
  switch (f) {                 // note fall-through!
    case 3: try { closeZ(); } catch (Exception e) {}
    case 2: if (didY) { try { closeY(); } catch … }
    case 1: try { closeX(); } catch (Exception e) {}
  }
}
```

## New Feature Motivation

- Avoid forgetting obligations
- No static program restrictions
- Optional lexical scoping
- Optional early or arbitrary cleanup
- Database / Workflow notions:
  - Either my actions all succeed (a1 a2 a3)
  - Or they rollback (a1 a2 error c2 c1)
  - Compensating transaction, linear saga

## Compensation Stacks

- Store cleanup code in run-time stacks
  - First-class objects, pass them around
- After “action” succeeds, push “cleanup”
  - “action” and “cleanup” are arbitrary code (anonymous functions)
- Pop all cleanup code and run it (LIFO)
  - When the stack goes out of scope
  - At an uncaught exception
  - Early, or when the stack is finalized

## Compensation Concepts

- Generalized destructors
  - No made-up classes for local cleanup
  - Can be called early, automatic bookkeeping
  - Can have multiple stacks
    - e.g., one for each request in a webserver
- Annotate interfaces to require them
  - Cannot make a new socket without putting “this.close()” on a stack of obligations
- Will be remembered along all paths
  - Details elsewhere …

## Cinderella Story

```
Stream input = new Stream();
Stream output = new Stream();
while (data = input.read())
    output.write(data);
output.close();
intput.close();
```

## “Assembly Language”

```
CompStack CS = new CompStack();
try {
  Stream input, output;
  compensate  { input = new Stream(); }
  with (CS)   { input.close(); }
  compensate  { output = new Stream(); }
  with (CS)   { output.close(); }
  while (data = input.read())
        output.write(data);
} finally { CS.runAll(); }
```

## With Annoted Interfaces

```
CompStack CS = new CompStack();
try {
  Stream input = new Stream(CS);
  Stream output = new Stream(CS);
  while (data = input.read())
        output.write(data);
} finally { CS.runAll(); }
```

## Using Most Recent Stack

```
CompStack CS = new CompStack();
try {
  Stream input = new Stream();
  Stream output = new Stream();
  while (data = input.read())
       output.write(data);
} finally { CS.runAll(); }
```

## Using Current Scope Stack

```
Stream input = new Stream();
Stream output = new Stream();
while (data = input.read())
       output.write(data);
```

## Cinderella 2 (Before)

```
int f = 0;                    // flag tracks progress
try {
  openX(); f = 1; work();
  if (…) { didY = true; openY(); f = 2; } work();
  openZ(); f = 3; work();
} finally {
  switch (f) {                // note fall-through!
    case 3: try { closeZ(); } catch (Exception e) {}
    case 2: if (didY) { try { closeY(); } catch … }
    case 1: try { closeX(); } catch (Exception e) {}
  }
}
```

## Cinderella 2 (After)

```
compensate          { openX(); }
with                { closeX(); }
work();
if (…) compensate   { openY(); }
       with         { closeY(); }
work();
compensate          { openZ(); }
with                { closeZ(); }
work();
// using the “current scope stack” by default
```

## Modeling Language

Each compensation stack comes with a unique name “i” based on its allocation site.

e ::= skip

| $e_1$ ; $e_2$

| if ★ then $e_1$ else $e_2$

| while ★ do e

| let $c_i$ = new CS() in e [ + $CS_i$ ]

| compensate $a_j$ with $b_j$ using $c_i$

| store $c_i$ [ - $CS_i$ ]

| let $c_i$ = load in e

| run $c_i$ [ - $CS_i$ ]

| runEarly $a_j$ from $c_i$

## Typing Judgment

- Live stack = stack that *may have* un-run compensating actions
- Dead stack = stack that *definitely has no* un-run compensating actions
- Judgment:

$$C, D \vdash e : C', D'$$

- Expression e typechecks in the context of live compensation stacks C and unused (dead) compensation stacks D and after executing e the new set of live stacks is C’ and the new set of dead stacks is D’

## Typing Rules

$$\frac{}{C, D \vdash \text{skip} : C, D}$$

$$\frac{C, D \vdash e_1 : C_1, D_1 \qquad C_1, D_1 \vdash e_2 : C_2, D_2}{C, D \vdash e_1 \,;\, e_2 : C_2, D_2}$$

$$\frac{C, D \vdash e_1 : C_1, D_1 \qquad C, D \vdash e_2 : C_2, D_2}{C, D \vdash \text{if } \bigstar \text{ then } e_1 \text{ else } e_2 : C_1 \cup C_2, D_1 \cap D_2}$$

$$\frac{C_1, D_1 \vdash e_1 : C_2, D_2 \qquad C_1 \cup D_1 = C_2 \cup D_2}{C_1, D_1 \vdash \text{while } \bigstar \text{ do } e : C \cup C_1, D \cap D_1}$$

## More Typing Rules

$$\frac{C_1, D_1 \cup \{i\} \vdash e : C_2, D_2 \qquad D_3 = D_2 \setminus \{i\}}{C_1, D_1 \vdash \text{let } c_i = \text{new CS() in } e : C_2, D_3}$$

$$\frac{i \in C}{C, D \vdash \text{compensate } a_j \text{ with } b_j \text{ using } c_i : C, D}$$

Not syntax-directed!
Why is this OK?

$$\frac{D_2 = D \setminus \{i\}}{C, D \vdash \text{compensate } a_j \text{ with } b_j \text{ using } c_i : C \cup \{i\}, D_2}$$

## Most Typing Rules

$$\frac{i \in C \cup D}{C, D \vdash \text{runEarly } a_j \text{ from } c_i : C, D}$$

$$\frac{C_2 = C \setminus \{i\}}{C, D \vdash \text{run } c_i : C_2, D \cup \{i\}} \qquad \frac{i \in D}{C, D \vdash \text{run } c_i : C, D}$$

Load is similar to “let/new”

Store is similar to “run”

## Case Studies

- Extend Java with compensation stacks
- Annotate key interfaces (File, DB, …)
- Annotate existing programs to use compensation stacks
  - For library resources
  - And for unique cleanup actions
  - No new exception handlers!
- Two studies: expressiveness, reliability

## Brown’s undo

- Provides operator-level time travel
  - Networked, logging SMTP and IMAP proxy
- 35,412 lines of Java, 128 change sites
  - Five- and three-step sagas
  - Complicated, unique cleanups with their own exception handling and synchronization
- Results
  - 225 lines shorter (~1%)
  - No measurable perf cost (1/2 std dev)

## Sun’s petstore

- “Amazon.com lite” plus inventory
  - Raises 150 exceptions over 3,900 requests
  - Avg Response: 52.06ms (std dev 100ms)
- 34,608 lines of Java, 123 change sites
  - Two hours of work
  - Three simultaneous resources (DB)
- Results:
  - 168 lines shorter (~0.5%)
  - 0 such exceptions over 3,900 requests
  - Avg Response: 43.44ms (std dev 77ms)

# Compensation Conclusions

- Combines static and dynamic analyses
  - CompStacks are tracked statically
  - Individual obligations are handled dynamically
- Easy to use for real-world programs
- Related to linear type systems

- Meh, seems to work.

# Homework

- Project Due
  - Need help? Stop by my office or send email.