Type Systems for Resource Management

Take-Home Message

• Type systems can be used to track how programs use 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 undecidable.
• 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: \( S_1 \vdash \text{cmd} : S_2 \)
  - \( S \) is the current set of resources
  - Linear type systems are flow-sensitive
  - Linear type systems behave like opsem
• Rules:

  \[
  \begin{align*}
  \{\text{Name}\} \in S_1 & \quad S_2 = S_1 \cup \{\text{Name}\} \\
  S_1 \vdash \text{new Name}, i : S_2 \\
  S_1 \vdash \text{cmd} : S_2 \\
  S_1 \vdash \text{del Name} : S_2 \\
  S_1 \vdash \text{cmd} : S_2 \\
  S_2 \vdash \text{cmd} : S_3 \\
  S_2 = S_1 \setminus \{\text{Name}\}
  \end{align*}
  \]

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.”

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

Are we obeying the rules?

Vault Intuition

- 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) [x(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

```c
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(...);
  bind(s, ...);
  listen(s, ...);
  while(true) {
    tracked sock t = accept(s);
    receive(t, buf);
    close(t);
  }
  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**

```c
void work() {
  if (p) {
  tracked sock s = socket(...);
  else
    skip;
  printf("hello world\n");
}
void aie() {
  DoublyLinkedList * L = NULL;
  while (rand() % 100 > 50) {
    tracked sock s = socket(...);
    L = PrependNode(s,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.
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

Error-Handling Errors

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 {
    Stream input = new Stream();
    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) { }
}

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

```
start
  new socket { socket }
  send { socket }
  close { socket }
  end
```

### Analysis Example

```
start
  new socket { socket }
  send { socket }
  close { socket }
  end
```

### Analysis Results

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

### 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

```java
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!)

```java
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 (Exception e) {} 
    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

**Assembly Language**

```java
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 Annotated Interfaces**

```java
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

```
e ::= skip  
| e₁ ; e₂  
| if ⋁ then e₁ else e₂  
| while ⋁ do e  
| let ci = new CS() in e  
| compensate aj with bj using ci  
| store ci  
| let ci = load in e  
| run ci  
| runEarly aj from ci
```

Typing Judgment

- **Live stack** = stack that *may have* un-run compensating actions
- **Dead stack** = stack that *definitely has no* un-run compensating actions
- **Judgment**:  
  \[ C, D ⊨ e : C', D' \]

- 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

\[
\begin{align*}
C, D & \\ C, D & \vdash \text{skip} : C, D \\
C, D & \vdash e_1 : C_1, D_1 \\
C, D & \vdash e_2 : C_2, D_2
\end{align*}
\]

\[
C, D \\ C, D \vdash e_1 : C_1, D_1, C_1, D_1 \vdash e_2 : C_2, D_2
\]

\[
\begin{align*}
C, D & \vdash \text{if } F \text{ then } e_1 \text{ else } e_2 : C_1 \cup C_2, D_1 \cap D_2 \\
C, D \vdash e_1 : C_1, D_1 \\
C, D \vdash e_2 : C_2, D_2
\end{align*}
\]

\[
\begin{align*}
C, D & \vdash \text{while } F \text{ do } e : C_1 \cup C_2, D_1 \cap D_2 \\
C, D \vdash e_1 : C_1, D_1 \\
C, D \vdash e_2 : C_2, D_2, D_1 \cup D_2
\end{align*}
\]

More Typing Rules

\[
\begin{align*}
C_1, D_1 \cup \{i\} & \vdash e : C_2, D_2 \\
C_1, D_1 & \vdash \text{let } c_i = \text{new CS()} \text{ in } e : C_2, D_2
\end{align*}
\]

\[
C_1, D_1 \vdash \text{compensate } a_j \text{ with } b_j \text{ using } c_i : C, D
\]

\[
\begin{align*}
i \in C \\
C, D \vdash \text{compensate } a_j \text{ with } b_j \text{ using } c_i : C \cup \{i\}, D_2
\end{align*}
\]

Not syntax-directed! Why is this OK?

Most Typing Rules

\[
\begin{align*}
i \in C \cup D \\
C, D & \vdash \text{runEarly } a_j \text{ from } c_i : C, D
\end{align*}
\]

\[
\begin{align*}
C_2 & = C \setminus \{i\} \\
\vdash C, D \vdash \text{run } c_i : C_2, D \cup \{i\} \\
C, D & \vdash \text{run } c_i : C, D
\end{align*}
\]

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%)
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.