CSCI 334: Principles of Programming Languages

Lecture 10: Control Structures I

Instructor: Dan Barowy Williams

Announcements

Graded HW2 back (except late assignments)

Announcements

Graded HW3 back this week
Announcements

Graded HW3 back this week
Hopefully HW4 early next week

Midterm Exam: Thurs, March 15
Mitchell Ch. 1-7
All material covered on homework

Midterm Exam Review:
Outside of class
Probably Tuesday, March 13 at 4pm
Location TBD

HW5 last homework before Spring Break.
Homework help session:
Thursday, March 8, 7-9 in TCL 206
No homework help session:
Thursday, March 15
Type Inference

Clever Uses for Types

Processor Clock Speeds


2,300 transistors  50,000 transistors  50,000,000 transistors


1000 MHz  1 MHz  10 MHz  100 MHz
Intel Core i7 (2010)

2,000,000,000 transistors

Multi-Core Chips

Core #1

Core #2

Core #3

Core #4

Concurrent Programming With Threads

Amazon.com

Network

Thread 1

Thread 2

Thread 3

Thread 4

Data
Multithreaded Program Execution

Thread A
... 
  t1 = bal; 
  bal = t1 + 100; 
  ... 

Thread B
... 
  t2 = bal; 
  bal = t2 - 100; 
  ... 

Race Condition

Thread A
... 
  t1 = bal; 
  bal = t1 + 100; 
  ... 

Thread B
... 
  t2 = bal; 
  bal = t2 - 100; 
  ...
Race Condition

A *race condition* occurs when two or more concurrent threads:
1. access the same variable at the same time,
2. at least one of the threads performs a write to the variable, and
3. the order of read/write events can cause the program to compute different results.

Avoiding Race Conditions

A *mutual exclusion barrier* (or *mutex*, or *lock*) is a concurrency control structure that prevents race conditions by limiting the possible thread interleavings.

**Thread Interleavings**

- **Thread A**
  - acquire(m);
  - \( t_1 = \text{bal}; \)
  - \( \text{bal} = t_1 + 100; \)
  - release(m);

- **Thread B**
  - acquire(m);
  - \( t_2 = \text{bal}; \)
  - \( \text{bal} = t_2 - 100; \)
  - release(m);

- **Thread A**
  - acquire(m);
  - \( t_1 = \text{bal}; \)
  - \( \text{bal} = t_1 + 100; \)
  - release(m);

- **Thread B**
  - acquire(m);
  - \( t_2 = \text{bal}; \)
  - \( \text{bal} = t_2 - 100; \)
  - release(m);

**Bal is 500**
Lock downside

Unfortunately, locks are not automatic. Programmers must identify the regions of code that must be protected (called a critical section) and manually insert locks.

Mutex Programming Bug

Thread A
synchronized(m) {
    t1 = bal;
    bal = t1 + 100;
}

Thread B
synchronized(m) {
    t2 = bal;
    bal = t2 - 100;
}

Type Inference to Identify Races

Thread 1
synchronized(l) {
    x = 10;
}
synchronized(m) {
    synchronized(l) {
        x = y + 1;
    }
    y = 2;
}

Thread 2
synchronized(m) {
    print y;
}
synchronized(m) {
    synchronized(l) {
        print x;
    }
}
Lock Type Inference Steps

1. Get abstract syntax tree
2. Label nodes with type labels
3. Generate constraints
4. Solve constraints
5. Type check: check that use is consistent

Lock Type Inference Constraints

\[ \begin{align*}
    \text{sync:} & \quad l:T, e:S \\
    T &= R \\
    S &= R \cup \{l\} \\
    \text{root:} & \quad R = \emptyset \\
    \text{other:} & \quad e1:T, e2:S \\
    \text{lockset:} & \quad i:V, lock_i \in V
\end{align*} \]

Infer Type

\[
\text{synchronized(l) \{}
    x = 10;
\}
\]

\[\begin{align*}
    a &= \emptyset \\
    b &= a \\
    c &= a \cup \{l\} \\
    d &= c \\
    e &= c \\
    \text{lock}_x \in d
\end{align*}\]

\[\text{x has lock type } 1\]

\[\text{i.e., } x \text{ is guarded by lock } 1\]
Activity

What are lock types for \( x \) and \( y \)?
Are they consistent with previous example?

```java
synchronized(m) {
    synchronized(l) {
        x = y + 1;
    }
    y = 2;
}
```

Consistency

```java
synchronized(l) {
    x = 10;
}
synchronized(m) {
    synchronized(l) {
        x = y + 1;
    }
    y = 2;
}
```

```
l_{lock x1} ∈ \{l\}
l_{lock x2} ∈ \{l,m\}
l_{lock y1} ∈ \{l,m\}
l_{lock y2} ∈ \{m\}
```

lock\(_{x1}\) ∩ lock\(_{x2}\) = \{l\}  lock\(_{y1}\) ∩ lock\(_{y2}\) = \{m\}

Lock set for variable should never be empty!

Type Inference to Identify Races

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Thread 1}</td>
<td>\text{Thread 2}</td>
</tr>
<tr>
<td>\text{synchronized(l) { \quad x = 10; }}</td>
<td>\text{synchronized(m) { \quad print y; }}</td>
</tr>
<tr>
<td>\text{synchronized(m) { \quad print x; }}</td>
<td>\text{synchronized(m) { \quad print x; }}</td>
</tr>
<tr>
<td>\text{synchronized(l) { \quad x = y + 1; }}</td>
<td>\text{synchronized(m) { \quad print y; }}</td>
</tr>
<tr>
<td>\text{y = 2;}</td>
<td>\text{y = 2;}</td>
</tr>
<tr>
<td>{l,m}</td>
<td>{l,m}</td>
</tr>
<tr>
<td>{l}</td>
<td>{m}</td>
</tr>
</tbody>
</table>

What's the problem here?

Call Stack

A \textit{call stack} is a control structure that stores information about the active subroutines of a program.

Most programming language runtimes use a call stack to evaluate a program instead of evaluation-by-substitution (i.e., \(\lambda\)-calculus reductions).
GC example from HW2

(car (cdr (cons (cons a b) (cons c b))))
GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))
  a b c d)

call stack

GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))
  a b c d)

call stack

GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))
  a b c d)

call stack

GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))
  a b c d)

call stack
GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))))

car x
cdr x
cons x
main x

call stack

GC example from HW2

(car (cdr (cons (cons a b) (cons c b)))))

car x
cdr x
cons x
main x

call stack
GC example from HW2

\((\text{car} \ (\text{cdr} \ (\text{cons} \ (\text{cons} \ a \ b) \ (\text{cons} \ c \ b)))))))\)

main \ x

call stack

Which objects are garbage?