Summer School on Language-Based Techniques for Concurrent and Distributed Software

Introduction

Dan Grossman
University of Washington
12 July 2006

Welcome!

1st of 32 lectures (4/day * 10 days = 32 ☺)
  – As an introduction, different than most

- A few minutes on the school, you, etc.
- A few minutes on why language-based concurrency
- Some lambda-calculus and naïve concurrency
- Rough overview of what the school will cover

I get 2 lectures next week on software transactions
  – Some of my research

A simple plan

- 11 speakers from 9 institutions
- “36” of you (28 PhD students, 5 faculty, 3 industry)
- Lectures at a PhD-course level
  - More tutorial/class than seminar or conference
  - Less homework and cohesion than a course
  - Not everything will fit everyone perfectly
    - Early stuff more theoretical
- Advice
  - Make the most of your time surrounded by great students and speakers
  - Be inquisitive and diligent
  - Have fun

Thanks!

- Jim: none of us would be here without him
- Jeff: the co-organizer
- Steering committee
  - Zena Ariola, David Walker, Steve Zdancewic
- Sponsors
  - Intel
  - National Science Foundation
  - Google
  - ACM SIGPLAN
  - Microsoft

Why concurrency

PL summer school not new; concurrency focus is

1. Concurrency/distributed programming now mainstream
   - Multicore
   - Internet
   - Not just scientific computing
2. And it’s really hard (much harder than sequential)
3. There is a lot of research (could be here 10 months)
4. A key role for PL to play…

Why PL

“what does it *mean* for computations to happen at the same time and/or in multiple locations”

“how can we best describe and reason about such computations”

Biased opinion: Those are PL questions and PL has the best intellectual tools to answer them

- “Learn concurrency in O/S class” a historical accident that will change soon

Why do people do it

If concurrent/distributed programming is so difficult, why do it?

- Performance
  (exploit more resources; reduce data movement)
- Natural code structure
  (independent communicating tasks)
- Failure isolation (task termination)
- Heterogeneous trust (no central authority)

It's not just "parallel speedup"

Outline

1. Lambda-calculus / operational semantics tutorial

2. Naively add threads and mutable shared-memory

3. Overview of the much cooler stuff we'll learn

"Starting with sequential" is only one approach

Remember this is just a tutorial/overview lecture

- No research results in the next hour

Lambda-calculus in *n* minutes

- To decide "what concurrency means" we must start somewhere
- One popular *sequential* place: a lambda-calculus
- Can define:
  - Syntax (abstract)
  - Semantics (operational, small-step, call-by-value)
  - A type system (filter out "bad" programs)

Syntax

Syntax of an *untyped lambda-calculus*

Expressions: `e ::= x | λx. e | e e |c | e+e`
"Constants: `c ::= … | -1 | 0 | 1 | …`"
"Variables: `x ::= x | y | x1 | y1| …`"
Values: `v ::= λx. e | c`

Defines a set of trees (ASTs)

Conventions for writing these trees as strings:

- `λx. e1 e2` is `λx. (e1 e2)`, not `(λx. e1) e2`
- *e1 e2 e3* is *(e1 e2) e3*, not *e1 (e2 e3)*
- Use parentheses to disambiguate or clarify

Semantics

- One computation step rewrites the program to something "closer to the answer"

$$e \rightarrow e'$$

- Inference rules describe what steps are allowed

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x.e)\ v \rightarrow e\{v/x\}}$$

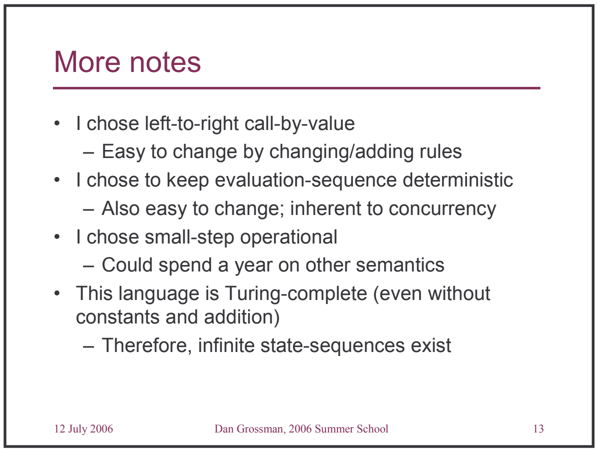
$$\frac{e1 \rightarrow e1'}{e1+e2 \rightarrow e1'+e2} \qquad \frac{e2 \rightarrow e2'}{v+e2 \rightarrow v+e2'} \qquad \frac{\text{“c1+c2=c3”}}{c1+c2 \rightarrow c3}$$

Notes

- These are rule schemas
  - *Instantiate* by replacing metavariables consistently
- A *derivation tree* justifies a step
  - A proof: "read from leaves to root"
  - An interpreter: "read from root to leaves"
- Proper definition of substitution requires care
- Program evaluation is then a sequence of steps

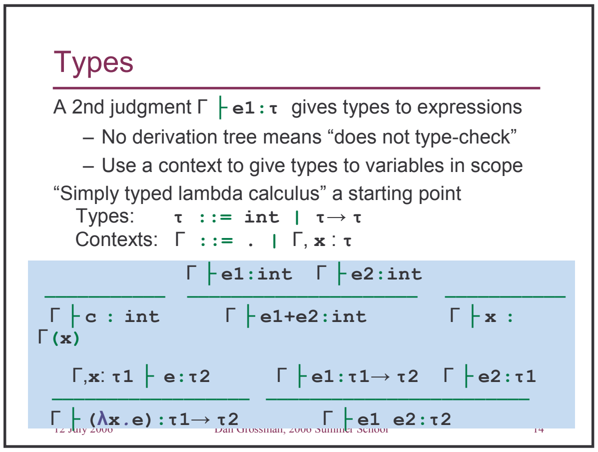
$$e0 \rightarrow e1 \rightarrow e2 \rightarrow \ldots$$

- Evaluation can "stop" with a value (e.g., 17) or a "stuck state" (e.g., 17 `λx. x`)



More notes

- I chose left-to-right call-by-value
  - – Easy to change by changing/adding rules
- I chose to keep evaluation-sequence deterministic
  - – Also easy to change; inherent to concurrency
- I chose small-step operational
  - – Could spend a year on other semantics
- This language is Turing-complete (even without constants and addition)
  - – Therefore, infinite state-sequences exist



Types

A 2nd judgment Γ ├ **e1:τ** gives types to expressions

- – No derivation tree means “does not type-check”
- – Use a context to give types to variables in scope

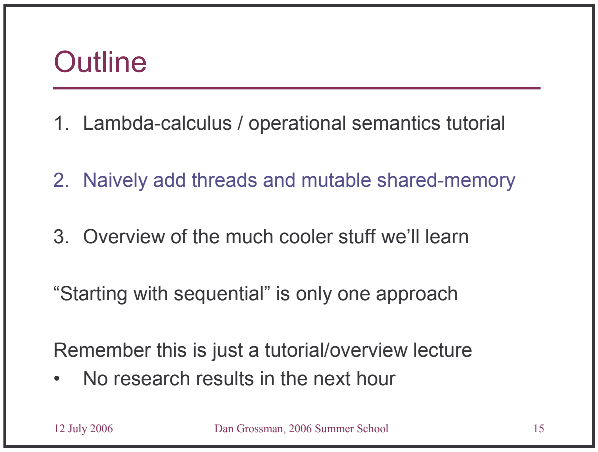
“Simply typed lambda calculus” a starting point

Types: **τ ::= int | τ→ τ**

Contexts: Γ **::= . |** Γ, **x** : **τ**

$$\frac{}{\Gamma \vdash \mathtt{c : int}} \qquad \frac{\Gamma \vdash \mathtt{e1:int} \quad \Gamma \vdash \mathtt{e2:int}}{\Gamma \vdash \mathtt{e1+e2:int}} \qquad \frac{}{\Gamma \vdash \mathtt{x :} \; \Gamma\mathtt{(x)}}$$

$$\frac{\Gamma,\mathtt{x}{:}\, \tau 1 \vdash \mathtt{e}:\tau 2}{\Gamma \vdash (\lambda \mathtt{x.e}):\tau 1 \to \tau 2} \qquad \frac{\Gamma \vdash \mathtt{e1}:\tau 1 \to \tau 2 \quad \Gamma \vdash \mathtt{e2}:\tau 1}{\Gamma \vdash \mathtt{e1\ e2}:\tau 2}$$



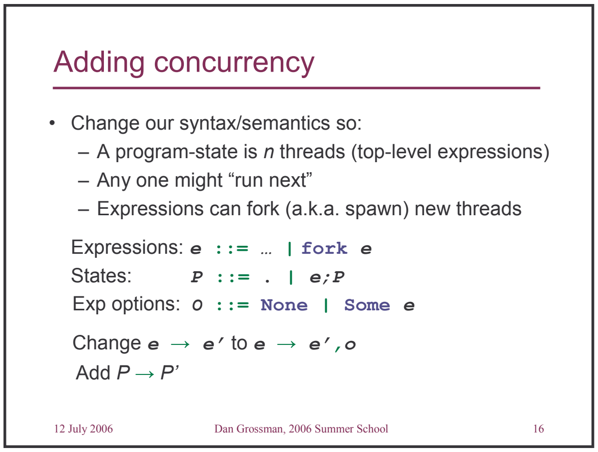
Outline

1. Lambda-calculus / operational semantics tutorial
2. Naively add threads and mutable shared-memory
3. Overview of the much cooler stuff we’ll learn

“Starting with sequential” is only one approach

Remember this is just a tutorial/overview lecture

- No research results in the next hour



Adding concurrency

- Change our syntax/semantics so:
  - – A program-state is *n* threads (top-level expressions)
  - – Any one might “run next”
  - – Expressions can fork (a.k.a. spawn) new threads

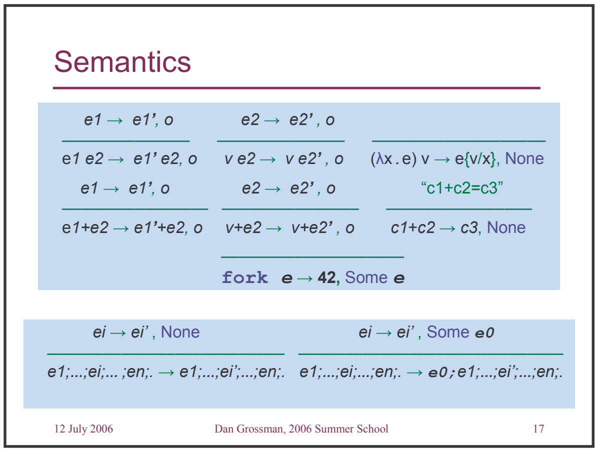
Expressions: ***e*** **::=** … **| fork *e***

States: ***P*** **::= . | *e*;*P***

Exp options: ***o*** **::= None | Some *e***

Change ***e*** → ***e’*** to ***e*** → ***e’*,*o***

Add *P* → *P’*



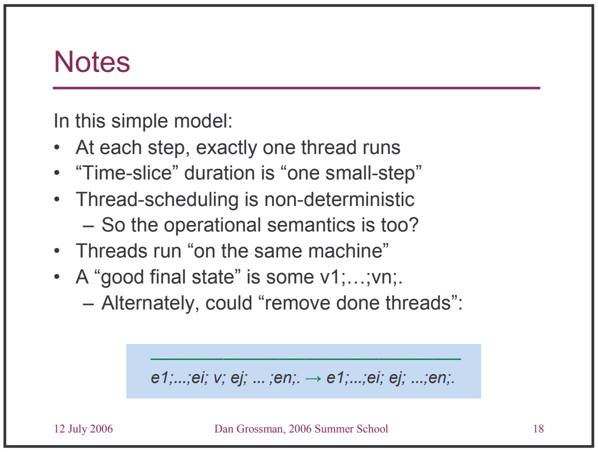
Semantics

$$\frac{e1 \to e1', o}{e1\ e2 \to e1'\ e2, o} \qquad \frac{e2 \to e2', o}{v\ e2 \to v\ e2', o} \qquad \frac{}{(\lambda x.e)\ v \to e\{v/x\}, \text{None}}$$

$$\frac{e1 \to e1', o}{e1{+}e2 \to e1'{+}e2, o} \qquad \frac{e2 \to e2', o}{v{+}e2 \to v{+}e2', o} \qquad \frac{\text{“c1+c2=c3”}}{c1{+}c2 \to c3, \text{None}}$$

$$\frac{}{\mathtt{fork}\ e \to \mathbf{42}, \text{Some}\ e}$$

$$\frac{ei \to ei', \text{None}}{e1;\ldots;ei;\ldots;en;. \to e1;\ldots;ei';\ldots;en;.} \qquad \frac{ei \to ei', \text{Some}\ e0}{e1;\ldots;ei;\ldots;en;. \to e0;e1;\ldots;ei';\ldots;en;.}$$



Notes

In this simple model:

- At each step, exactly one thread runs
- “Time-slice” duration is “one small-step”
- Thread-scheduling is non-deterministic
  - – So the operational semantics is too?
- Threads run “on the same machine”
- A “good final state” is some v1;…;vn;.
  - – Alternately, could “remove done threads”:

$$\frac{}{e1;\ldots;ei;\ v;\ ej;\ \ldots;en;. \to e1;\ldots;ei;\ ej;\ \ldots;en;.}$$

Not enough

- These threads are really uninteresting; they can’t communicate
  - One thread’s steps can’t affect another
  - All final states have the same values
- One way: *mutable shared memory*
  - Many other communication mechanisms to come!
- Need:
  - Expressions to create, access, modify mutable locations
  - A map from mutable locations to values in our program state

Changes to old stuff

Expressions: `e ::= …| ref e |e1 := e2 | !e | l`
Values: `v ::= …| l`
Heaps: `H ::= . | H,l→v`
Thread pools: `P ::= . | e;P`
States: `H,P`

Change $e \rightarrow e', o$ to $H, e \rightarrow H', e', o$
Change $P \rightarrow P'$ to $H, P \rightarrow H', P'$
Change rules to modify heap (or not). 2 examples:

$$\frac{H, e1 \rightarrow H', e1', o}{H, e1\ e2 \rightarrow H', e1'\ e2, o} \qquad \frac{\text{“c1+c2=c3”}}{H, c1+c2 \rightarrow H, c3, \text{None}}$$

New rules

$$\frac{\texttt{l} \text{ not in } H}{H, \texttt{ref}\ v \rightarrow H, \texttt{l}\rightarrow \texttt{v}, \texttt{l}, \text{None}} \qquad \frac{}{H, \texttt{!}\ \texttt{l} \rightarrow H, H(\texttt{l}), \text{None}}$$

$$\frac{}{H, \texttt{l} := \texttt{v} \rightarrow H, \texttt{l}\rightarrow \texttt{v}, \texttt{42}, \text{None}}$$

$$\frac{H, e \rightarrow H', e', o}{H, \texttt{!}\ e \rightarrow H', \texttt{!}\ e', o} \qquad \frac{H, e \rightarrow H', e', o}{H, \texttt{ref}\ e \rightarrow H', \texttt{ref}\ e', o}$$

$$\frac{H, e \rightarrow H', e', o}{H, e1 := e2 \rightarrow H', e1' := e2, o} \qquad \frac{H, e \rightarrow H', e', o}{H, v := e2 \rightarrow H', v := e2', o}$$

Now we can do stuff

We could now write “interesting examples” like

- Fork 10 threads, each to do a different computation
- Have each add its answer to an accumulator `l`
- When all threads finish, `l` is the answer

Problems:

1. If this is not the whole program, how do you know when all 10 threads are done?
   - Solution: have them increment another counter
2. If each does `l := !l + e`, there are races…

Races

`l := !l + 35`

An interleaving that produces the wrong answer:

Thread 1 reads `l`
Thread 2 reads `l`
Thread 1 writes `l`
Thread 2 writes `l` – “forgets” thread 1’s addition

Communicating threads must synchronize

Languages provide synchronization mechanisms, e.g., locks…

Locks

Two new expression forms:

- acquire e
  if e is a location holding 0, make it hold 1
  (else *block*: no rule applies; thread temporarily stuck)
  (test-and-set is atomic)
- release e
  same as e := 0; added for symmetry

Adding formal inference rules: “exercise”
Using this for our example: “exercise”
Adding *condition variables*: “more involved exercise”

Locks are hard

Locks can avoid races *when properly used*

- But it's up to the programmer
- And "application-level races" may involve multiple locations
  - Example: "`l1 > 0` only if `l2 = 17`"

Locks can lead to deadlock

Trivial example:

acquire l1	acquire l2
acquire l2	acquire l1
release l2	release l1
release l1	release l2

Summary

We added

1. Concurrency via fork and non-deterministic scheduling
2. Communication via mutable shared memory
3. Synchronization via locking

There are better models; this was almost a "straw man"

Even simple concurrent programs are hard to get right

- Races and deadlocks common

And this model is much simpler than reality

- Distributed computing; relaxed memory models

Outline

1. Lambda-calculus / operational semantics tutorial
2. Naively add threads and mutable shared-memory
3. Overview of the much cooler stuff we'll learn

"Starting with sequential" is only one approach

Remember this is just a tutorial/overview lecture

- No research results in the next hour

Some of what you will see

1. Richer foundations (theoretical models)
2. Dealing with more complicated realities
3. Other communication/synchronization primitives
4. Techniques for improving lock-based programming

[This is not in the order we will see it]

Foundations

- Process-calculi [Sewell]
  - Inherently parallel (rather than an add-on)
  - Communication over channels
- Modal logic [Harper]
  - Non-uniform resources
  - Types for distributed computation
- Provably efficient job scheduling [Leiserson/Kuszmaul]
  - Optimal algorithms for load-balancing

Realities

- Distributed programming [Sewell] [Harper]
  - Long latency, lost messages, version mismatch, …
- Relaxed memory models [Dwarkadas]
  - Hardware does not give globally consistent memory
- Dynamic software updating [Hicks]
  - Cannot assume fixed code during execution
- Termination [Flatt]
  - Threads may be killed at inopportune moments

Ways to synchronize, communicate

- Fork-join [Leiserson/Kuszmaul]
  - Block until another computation completes
- Futures [Hicks]
  - Asynchronous calls (less structured fork/join)
- Message-passing a la Concurrent ML [Flatt]
  - First-class synchronization events to build up communication protocols
- Software transactions, a.k.a. atomicity…

Atomicity

An easier-to-use and harder-to-implement synchronization primitive:

`atomic { s }`

Must execute s *as though* no interleaving, but still ensure *fairness*.

- Language design & software-implementation issues [Grossman]
- Low-level software & hardware support [Dwarkadas]
- As a checked/inferred annotation for lock-based code [Flanagan]

Analyzing lock-based code

- Type systems for data-race and atomicity detection [Flanagan]
  - Static & dynamic enforcement of locking protocols
- Analysis for multithreaded C code; “what locks what” [Foster]
  - Application to systems code; incorporating alias analysis
- Model-checking concurrent software [Qadeer]
  - Systematic state-space exploration

Some of what you will see

1. Richer foundations (theoretical models)
2. Dealing with more complicated realities
3. Other communication/synchronization primitives
4. Techniques for improving lock-based programming

[This is not in the order we will see it]

Thanks in advance for a great summer school!