Logistics/Reminders

- HW6 due tonight
- Course eval open through Sunday
  - Bonus points for everyone: $2 \times (\text{response rate})^2$
- Review session: Monday 11-12, SB 106 (and Zoom and recorded)
- Final: **Tuesday, May 2, 10:30am-12:30pm, SB 104**
Content

• Simple answer: everything!
• All lectures, from the beginning of the semester until this Thursday
  • More emphasis on material since midterm
  • Only high-level questions about post-HW6 material
• Written questions from HW5, HW6 and the midterm are good examples of the types of questions I might ask
Format

• 120 minutes, 100 points

• Approx. 50%:
  • A few short answer questions
  • Give the value of an OCaml expression or say it doesn’t evaluate (like on midterm)
  • Write a proof tree for a big-step semantics or typing derivation (like HW5, 2.1 and HW6, 1.2)
  • Evaluate a lambda calculus term to a normal form (like HW5, 3.3, but you only have to do one)

• Approx. 50%: 2-3 more long questions
Rules, etc.

• Write in whatever you want (please no red/green/purple pen though)

• You can bring two double-sided 8.5x11” sheets of notes
  • Written or typed, can contain anything you want
  • One can be the one from the midterm

• Provided reference material (I will give this to you at the exam, no need to print it or put it on your note sheets):
  • Signatures for OCaml list functions
  • IMP syntax and big-step rules
  • STLC syntax and typing rules
Practice, review

• Practice exam posted on Blackboard today or tomorrow, with reference material
  • Same basic format as real exam, but I make no promises about exact difficulty, length

• Review session
  • Monday, 5/1 11am-12pm (instead of office hours)
  • SB 106
  • Will also be streamed and recorded – I’ll send out the link
  • Come with questions!
Schedule

• Intro (1 week)
• Learn OCaml (~4 weeks)
• Interpreters (~2 weeks)
• Midterm
• Type checking (~2 weeks)
• Spring break
• Formal semantics (~2 weeks)
• Formal type systems (~2 weeks)
• Other topics and wrap-up (~3 weeks)

Programming Languages
Implementing PLs
Reasoning about PLs
Knowing the right paradigm to use can make programming easier

Task: Sort a linked list (using merge sort)

Try writing even a minimal working web server in C in an hour!
Knowing about the language and how it’s translated can help you write faster code.

Merge sort, 10,000 elements

Time (ms)

- C: 7
- Python: 6430
- OCaml (bytecode): 14
- OCaml (native): 10
Type systems can express different levels of guarantees

- **C**
  
  \[ \text{node *} \text{mergesort} (\text{node *} \text{list}) \]
  
  Takes a pointer to a node and returns a pointer to a node.

- **OCaml**
  
  \[ \text{mergesort} : \text{int list} -\rightarrow \text{int list} \]
  
  Takes an integer list and returns an integer list.

- **Haskell**
  
  \[ \text{mergesort} :: \text{IO} ([\text{int}] -\rightarrow [\text{int}]) \]
  
  Takes an integer list, returns an integer list and performs I/O (e.g., printing).

- **Coq**
  
  \[ \text{mergesort} : \forall (l1 : \text{list int}), \exists (l2 : \text{int list}), \]
  
  \[
  \text{Sorted } l2 \text{ } /\text{ }\backslash \text{ } \text{Permutation } l1 \text{ } l2
  \]
  
  Takes an integer list and returns a sorted permutation of it.
Different languages are up to different tasks

OCaml

C?

Rust?
Schedule

• Intro (1 week)
• Learn OCaml (~4 weeks)
• Interpreters (~2 weeks)
• Midterm
• Type checking (~2 weeks)
• Spring break
• Formal semantics (~2 weeks)
• Formal type systems (~2 weeks)
• Other topics and wrap-up (~3 weeks)

} Programming Languages
| Implementing PLs
| Reasoning about PLs
Compilers vs. interpreters

• Compiler
  • Translates the program to a form executable by the machine (or assembly)
  • Compile, then can run the executable: compiler no longer involved

• Interpreter
  • Doesn’t translate to machine-readable format
    • Might compile to bytecode or intermediate representation
  • Runs (“interprets”) program directly
  • Can’t run without the interpreter
Compilers translate code in phases


a = b + c - 1

VAR a EQUAL
VAR b OP +
VAR C OP - CONST 1

Assign
a + b - c 1

temp = c - 1
a = b + temp

subl %rax, 1
addl %rax, %rbx

"Front End" → Analysis → "Back End"
Compiler collections also swap out front ends for different languages
Want to see more?
Take CS443 (Compiler Construction)
Schedule

- Intro (1 week)
- Learn OCaml (~4 weeks)
- Interpreters (~2 weeks)
- **Midterm**
- Type checking (~2 weeks)
- Spring break
- Formal semantics (~2 weeks)
- Formal type systems (~2 weeks)
- Other topics and wrap-up (~3 weeks)
Type safety: well-typed programs don’t “go wrong”

- Progress: A well-typed program isn’t wrong (in STLC: stuck)
- Preservation: If a well-typed program takes a step, it’s still well-typed
“Go wrong” can mean lots of other things

- One application we haven’t talked (much) about: parallelism
Functional languages are **great** for parallelism

```plaintext
let (a, b) = (f (), g ())
```

- If f and g are functional, it can’t matter what order we execute them in...
- so why not do them at the same time?

- Deadlocks
- Locking
- Data races
Cilk, Go, Parallel ML, Parallel Haskell, ...

Theorem: \( T(P) \leq \frac{W}{P} + S \)

So why don’t people use the abstractions?

(Short answer: user interaction)

POSIX threads
Use multiple threads to do a lot of things
With parallelism, stops being responsive

Many lightweight threads running AI
Some tasks have higher priority than others
Simple priority syntax

```r
priority sensors
priority short_term_planning
priority long_term_planning
order long_term_planning < short_term_planning
order short_term_planning < sensors

sensethread <- spawn[sensors] { ... };
plan1 <- spawn[short_term_planning] { ... };
plan2 <- spawn[long_term_planning] { ... };
```
The program went wrong

• How do we stop programs from going wrong?
We track priorities through code in **types**

order low < high

cmd[high]
{
  t <- spawn[high] { … };
  ...
  sync(t)
}

• This thread is high-priority
• Spawn a high-priority thread
• Sync on it

.constraint violated at example.prm:5.1-5.8 : high <= low
Type error: constraint violated
We want to sync on \( e \) from priority \( \rho' \).

\[ \Gamma \vdash e : \tau \ \text{thread}[\rho] \quad \rho \geq \rho' \]

\[ \Gamma \vdash \text{sync}(e) : \tau @ \rho' \]

\( e \) is a handle to a thread of priority \( \rho \).

\( e \) is higher priority than the current thread.
What if a thread wants to change its priority?

priority sensors
priority short_term_planning
priority long_term_planning
order long_term_planning < short_term_planning
order short_term_planning < sensors

plan1 <- spawn[long_term_planning]
{ ...
  if time > deadline - 5ms then
    change[short_term_planning];
  ...
};

Extension to type system being done currently by a CS440 Spring 2021 student!
Want to learn

• How to prove progress and preservation?
• More advanced type systems that can express more complex programs?
• How to design new type systems for things you want to express about programs?

Take CS534 (Types and Programming Languages)
Hoare Logic can verify other properties

• Remember: $\vdash \{P\} S \{Q\}$
• “if P holds before and S terminates, Q holds after”

• $\vdash \{n \geq 0\} x := \text{fact}(n) \{x = n!\}$

• How do we prove this?
With inference rules!

\[
\text{SKIP} \Rightarrow \{P\} \text{skip} \{P\} \\
\text{ASSIGN} \Rightarrow \{[E/x] P\} x := E \{P\} \\
\Rightarrow \{((x + 1)/x)(x = 1)\} x := x + 1 \{x = 1\}
\]
With inference rules!

\[
\begin{align*}
\text{SEQ} & \quad \frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\Rightarrow \{P\} S_1; S_2\{R\}} \\
\text{WHILE} & \quad \frac{\{P \land B\} S \{P\}}{\Rightarrow \{P\} \text{while } B \text{ do } S \{P \land \neg B\}}
\end{align*}
\]

\[
\begin{align*}
\{n \geq 0\} x := 1; i := 2 \{x = (i - 1)!\} & \quad \ldots \\
\{x = (i - 1)! \land i \leq n\} (x := x \ast i; i := i + 1) \{x = (i - 1)!\} & \quad \ldots \\
\{x = (i - 1)!\} \text{while } i < n \text{ do } (x := x \ast i; i := i + 1) \{x = (i - 1)! \land i > n\} & \quad \ldots \\
\Rightarrow \{n \geq 0\} x := 1; i := 2; \text{while } i \leq n \text{ do } (x := x \ast i; i := i + 1) \{x = n!\}
\end{align*}
\]
method ComputeFib(n: nat) returns (r: nat)
ensures r == fib(n)
{
var a, b := 0, 1;
var temp := 0;
var i := 0;
while (i < n)
invariant (a == fib(i)) && (b == fib(i+1)) && (i <= n)
{
temp := a + b;
assert temp == fib(i+2);
    a := b;
    b := temp;
    i := i + 1;
}
return a;
Want to learn
• How to use Hoare Logic to prove real things about real programs?
• About total correctness (proving programs terminate)?
• About verifying concurrent programs?
Take CS536 (Science of Programming)
What to take next?

- Write a compiler!
- Coding or theory?
  - Theory!
  - Types?
    - All the types!
      - 443
    - No thanks
      - 536
      - 534

Like this stuff (especially the priority type system) and want to do it more hands-on? I’m looking for research assistants! Email me!