Building Interpreters: Recap

CS 440: Programming Languages
Stefan Muller
Slides largely by Michael Lee <lee@iit.edu>
HW2

- Due tonight, 11:59pm (can take <= 2 late days as usual)
- For hof.ml and trees.ml:
  - You may not write any recursive (including tail-recursive) functions, except on the bonus question (and copy/pasting tree_fold)
- For all parts:
  - You can use any operators or library functions we’ve seen, as long as it isn’t just what you’re supposed to implement.
  - Examples of what’s allowed: ^, @, List.init (will be very useful)
  - Not allowed: List.concat for implementing concatenate
Midterm: Thursday, 3/2

- In-class, 75 minutes
- Covers Lectures 0-13 (through today), Homeworks 0-2
Non-exhaustive list of topics

- Types of programming languages
- Interpreters vs. compilers
- Structure of an interpreter/compiler
- OCaml programming
  - Types, expressions, evaluation, (tail) recursion
  - Algebraic data types
  - Higher-order functions
- Interpreters
  - Environments
Format

- 4-5 (multi-part) questions
- Short answer, some small programming questions
Other info

- Write in blue or black pen only (no pencil)
- I reserve the right to deduct 5 points from exams written in pencil
- You can bring one double-sided 8.5x11” sheet of notes
- Written or typed, can contain anything you want
- I’ll give you type signatures for the usual HOFs
- Anything else you want? Let me know on Discord by tomorrow
Other info (continued)

- I’ll post a practice exam soon

- Instead of Thursday office hours next week, I’ll have a Zoom review session Wed., 3/1 11-12
Overview
“Traditional” Interpreter Workflow

Source code (Plain text) → Lexer → Tokens → Parser → Syntax tree

- Lexer
- Tokens
- Parser
- Syntax tree

Analysis / Optimization

- Analysis / Optimization
- IR
- Evaluator
Our Implementation

Source code (Plain text) -> Lexer -> Tokens -> Parser -> Syntax tree

Desugarer -> Syntax tree (core language) -> Evaluator

More on this in a bit!
Compilation Workflow

Source code (Plain text) -> Lexer -> Tokens -> Parser -> Syntax tree

Lexer
Parser
Syntax tree

Analysis / Optimization

IR

Code Generator

Bytecode / Machine code

Compilation Workflow
§ Some implementation details
Identifier bindings

- let and fun forms bind identifiers within specific scopes
- An expression’s *environment* comprises all bindings in effect when it is evaluated
Identifier bindings

- We use an association list to represent an environment

- E.g., [(x, ref 44); (y, ref 10)]

- *Immutable structure*: bindings are prepended when recursing

- Bindings may be mutably updated to allow backpatching
Identifier bindings

```
let x = 44 in
let y = 10 in
x * y
```

```
env=[]
eval

let x = 44 in
let y = 10 in
x * y
```

```
eval
env=[(x, ref 44)]
```

```
eval
env=[(y, ref 10); (x; ref 44)]
```
Identifier bindings

```plaintext
eval
env=[]
   let x = 44 in
   let y = 10 in
   let x = 54 in
   x * y
```

```
eval
env=[(x, ref 44)]
   let x = 54 in
   x * y
```

```
eval
env=[(x, ref 44)]
   let y = 10 in
   let x = 54 in
   x * y
```

```
eval
env=[(y, ref 10); (x, ref 44)]
   let x = 54 in
   x * y
```

```
eval
env=[(y, ref 10); (x, ref 44)]
   let x = 54 in
   x * y
```

```
eval
env=[(x, ref 54); (y, ref 10); (x, ref 44)]
   x * y
```

(shadowed)
let/lambda equivalence

- Note that all let forms can be written as lambda applications!

\[
\begin{align*}
\text{let } x = 44 \\
\text{in } x * 10
\end{align*}
\]
\[
\Leftrightarrow (\text{fun } x \rightarrow x * 10) \ 44
\]

\[
\begin{align*}
\text{let } x = 44 \text{ in } \\
\text{let } y = 3 + 7 \text{ in } \\
x * y
\end{align*}
\]
\[
\Leftrightarrow (\text{fun } x \ y \rightarrow x * y) \ 44 \ (3 + 7)
\]
Evaluation strategies

- Question: **when** do we evaluate expressions in binding forms?
  
  - E.g., let \( x = 1 + 2 \) in ...
    
    \((\text{fun } x \to \ldots)(1 + 2)\)
  
- Two general strategies: **Eager** and **Lazy**
Eager evaluation

- Evaluate *before* binding the identifier

- aka **call-by-value**: evaluated “value” is passed as arg to function
Lazy evaluation

- Evaluate the expression *only when needed*

- aka **call-by-name**: un-evaluated expression "name" is passed

- An efficient version may cache (memoize) evaluated results instead of re-evaluating

```
ext x = 1 + 2 in x + x + 4
(1 + 2) + (1 + 2) + 4
3 + (1 + 2) + 4
3 + 3 + 4
10
```

```
ext x = 1 + 2 in x + x + 4
(1 + 2) + (1 + 2) + 4
3 + 3 + 4
10
```
Eager vs. Lazy

- Eager evaluation is much more common in modern languages
  - More predictable behavior; easier to analyze program requirements
  - Often more efficient than a non-memoizing lazy evaluator
- Lazy evaluation may avoid doing unnecessary work (e.g., unreferenced identifiers in a function)
  - Control flow can be implemented via regular functions
- Infinite / partially defined data structures are easy to define
Control flow with functions

type my_bool = True | False

let my_if (e: my_bool) (if_b: ‘a) (else_b: ‘a) =
    match e with
    | True -> if_b
    | False -> else_b

my_if True (1 + 2) (42 / 0)

my_if True 3 !!!!!
Control flow with functions - Lazy

type my_bool = True | False

let my_if (e: my_bool) (if_b: 'a) (else_b: 'a) =
    match e with
    | True -> if_b
    | False -> else_b

my_if True (1 + 2) (42 / 0)

match True with True -> 1 + 2 | False -> 42 / 0

1 + 2
Scope selection

- Question: **which bindings** (for free variables) are used when evaluating a function (lambda)?

  - E.g.,
    
    ```
    let f = let x = 44 in
    fun y ->
    x * y
    in
    let x = 33 in
    f 10
    ```

- Two strategies: **Dynamic** and **Lexical**
Dynamic binding

- Use the scopes in effect where the function is **called**
- I.e., free variables are looked up in the *dynamic environment*

```
let f = let x = 5 in
    fun y -> x * y
in
  (let x = 4 in f 10) + (let x = 3 in f 10)
```

> 70
Lexical binding

- Use the scopes in effect where the function is **defined**
- I.e., a function captures or “closes over” bindings in its *lexical environment*
- Lexically bound functions = Closures

```ml
let f = let x = 5 in
  fun y -> x * y
in
  (let x = 4 in f 10)
+ (let x = 3 in f 10)
> 100
```
Closure implementation

- A closure couples a function with its lexical environment
- An efficient version would only keep required bindings
- Critical for languages with *first-class functions*
- Functions may outlive their defining environment, but need to hang onto bindings!
Desugaring

- Question: how to add syntactic elements (and associated semantics)?
- Option 1: update parser & evaluator — all syntax is first class
- Option 2: translate new syntactic elements into **core language**
  - Performed during “desugaring” passes (syntactic sugar → core syntax)
  - Keeps core language small and easy to reason about / test!
Desugaring

- E.g., `fun x y z -> body ...`

(can also desugar let -> application)

```
fun x ->
  fun y ->
    fun z -> body
```
Short-circuiting and/or

- if \( x > 0 \) \&\& \( y / x > 5 \) then 1 else 2

- Remember eval case for \( \text{EBinop} \) (\( e_1, o, e_2 \)):

\[
\text{let } v_1 = \text{eval_expr } e_1 \text{ env in}
\text{let } v_2 = \text{eval_expr } e_2 \text{ env in}
\text{eval_op } o \ v_1 \ v_2
\]

\[
\text{eval_expr } (\text{EBinop} \ (x > 0, \text{And}, \ y / x > 5)) \ ?
\]
Short-circuiting and/or

- if \( x > 0 \) && \( y / x > 5 \) then 1 else 2

- if if \( x > 0 \) then \( y / x > 5 \) else false then 1 else 2
What did we leave out?

- Parsing!
- Language independent intermediate representations (e.g., LLVM)
- Optimizations (e.g., lean/fast environments, efficient execution)
- Memory management
- Code generation (transpiling, bytecode/machine code generation)
- Take **CS 443: Compiler Construction**!