Interpreters

Nate Foster
Spring 2018
Review

Previously in 3110:
• functional programming
• modular programming
• data structures

Today:
• new unit of course: interpreters
• small-step interpreter for tiny language
COMPILERS AND INTERPRETERS
code as data: the compiler is code that operates on data; that data is itself code
Compilation

The diagram illustrates the process of compilation, where a source program is compiled by a compiler into a target program. The target program is then executed to produce output. The compiler goes away; it is not needed to run the program.
Interpretation

Source program

Input ➞ Interpreter ➞ Output

the interpreter stays; needed to run the program
Compilation vs. interpretation

• Compilers:
  – primary job is *translation*
  – typically lead to better performance of program

• Interpreters:
  – primary job is *execution*
  – typically lead to easier implementation of language
    • maybe better error messages and better debuggers
Mixed compilation and interpretation

the VM is the interpreter; needed to run the program; Java and OCaml can both work this way
Architecture

Two phases:

- **Front end**: translate source code into *abstract syntax tree* (AST)
- **Back end**: translate AST into machine code

Front end of compilers and interpreters largely the same:

- *Lexical analysis* with *lexer*
- *Syntactic analysis* with *parser*
- *Semantic analysis*
Front end

Character stream:
if x=0 then 1 else fact(x-1)

Token stream:
if x = 0 then 1 else fact (x - 1)
Front end

Token stream:

```plaintext
if x = 0 then 1 else fact (x - 1)
```

Abstract syntax tree:
Front end

Abstract syntax tree:

```
if-then-else
  = 1
  apply
    fact
    -
    x 1
```

Semantic analysis

- accept or reject program
- *decorate* AST with types
- etc.
After the front end

- **Interpreter** begins executing code using the abstract syntax tree (AST)
- **Compiler** begins translating code into machine language
  - Might involve translating AST into a simpler *intermediate representation* (IR)
  - Eventually produce *object code*
Functional languages are well-suited to implement compilers and interpreters

- **Code** easily represented by tree data types
- **Compilation** passes easily defined pattern matching on trees
- **Semantics** naturally implemented with language constructs
EXPRESSION INTERPRETER
Arithmetic expressions

Goal: write an interpreter for expressions involving integers and addition

Path to solution:
• let's assume lexing and parsing is already done
• need to take in AST and interpret it
• intuition:
  – an expression $e$ takes a single step to a new expression $e'$
  – expression keeps stepping until it reaches a value
type expr =
  | Int of int
  | Add of expr * expr

e.g.
• Int 5 represents the source expression 5
• Add (Int 5)
  (Add (Int 6) (Int 7))
represents 5+(6+7)
Evaluation by stepping

(* A single step of evaluation:
  * exactly 1 step *)
val step : expr -> expr

(* Take as many steps as possible until
  * a value is reached. Could be 0
  * or more steps. *)
val eval : expr -> expr
Eval

let rec eval e =
  if is_value e
  then e
  else eval (step e)

(* [is_value e] is whether * [e] is a syntactic value *)

let is_value = function
  | Int _ => true
  | Add _ => false
let rec step = function
    | Int n -> failwith "Does not step"
    | Add(e1, e2) -> Add(step e1, e2)
Step, Choice A

```
let rec step = function
| Int n -> failwith "Does not step"
| Add(e1, e2) -> Add(step e1, e2)
| Add(Int n1, e2) -> Add(Int n1, step e2)
```
Step, Choice A

let rec step = function
    | Int n -> failwith "Does not step"
    | Add(e1, e2) -> Add(step e1, e2)
    | Add(Int n1, e2) -> Add(Int n1, step e2)

Stop: we already have a bug

How will 5+(6+7) step?
let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, e2) -> Add(Int n1, step e2)
  | Add(e1, e2) -> Add(step e1, e2)
Step, Choice A

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1+n2)
  | Add(Int n1, e2) -> Add(Int n1, step e2)
  | Add(e1, e2) -> Add(step e1, e2)
let rec step = function

  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1+n2)
  | Add(Int n1, e2) -> Add(Int n1, step e2)
  | Add(e1, e2) -> Add(step e1, e2)

Finished!
let rec step = function
| Int n  -> failwith "Does not step"
| Add(Int n1, Int n2) -> Int (n1+n2)
| Add(e1, Int n2) -> Add(step e1, Int n2)
| Add(e1, e2) -> Add(e1, step e2)
EXTENDED EXPRESSION
INTERPRETER
Arithmetic expressions

Goal: extend interpreter to let expressions

Path to solution:
• extend AST with a variant for let and for variables
• add branches to step to handle those
• that requires substitution...
**let expressions [from lec 4]**

```
let x = e1 in e2
```

**Evaluation:**

- Evaluate `e1` to a value `v1`
- **Substitute** `v1` for `x` in `e2`, yielding a new expression `e2'`
- Evaluate `e2'` to `v`
- Result of evaluation is `v`
Substitution

• Notation: \( e\{v/x\} \) means \( e \) with \( v \) substituted for \( x \)
  – e.g., \( (x+5)\{4/x\} \) means \( (x+5) \) with 4 substituted for \( x \)
  – which would be \( (4+5) \)

• In let semantics:
  – Instead of: "Substitute \( v_1 \) for \( x \) in \( e_2 \), yielding a new expression \( e_2' \); Evaluate \( e_2' \) to \( v \)"
  – Could now write: "Evaluate \( e_2 \{v_1/x\} \) to \( v \)"
Extended AST

type expr =
  | Int of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr

e.g.
- Var "x" represents the source expression x
- Let "x" (Int 5) (Add (Var "x") (Int 1)) represents let x = 5 in x+1
Eval

let rec eval e =
    if is_value e then e
    else eval (step e)

let is_value = function
    | Int _  -> true
    | Add _  | Var _  | Let _  -> false
let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(Int n1, e2) -> Add (Int n1, step e2)
  | Add(e1, e2) -> Add (step e1, e2)
Step

\[
\text{let rec } \text{step } = \text{ function }
\]

\[\begin{align*}
| \text{Int } n & \rightarrow \text{ failwith } "\text{Does not step}" \\
| \text{Add(Int } n_1, \text{ Int } n_2) & \rightarrow \text{ Int } (n_1 + n_2) \\
| \text{Add(Int } n_1, \text{ e2}) & \rightarrow \text{ Add } (\text{Int } n_1, \text{ step } \text{ e2}) \\
| \text{Add(e1, e2)} & \rightarrow \text{ Add } (\text{step } e_1, \text{ e2}) \\
| \text{Var } _ & \rightarrow \text{ failwith } "\text{Unbound variable}" \\
\end{align*}\]

Why? Equivalent to just typing "x;;" into fresh utop session
let rec step = function
| Int n -> failwith "Does not step"
| Add(Int n1, Int n2) -> Int (n1 + n2)
| Add(Int n1, e2) -> Add (Int n1, step e2)
| Add(e1, e2) -> Add (step e1, e2)
| Var _ -> failwith "Unbound variable"
| Let(x, e1, e2) -> Let (x, step e1, e2)
let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(Int n1, e2) -> Add (Int n1, step e2)
  | Add(e1, e2) -> Add (step e1, e2)
  | Var _ -> failwith "Unbound variable"
  | Let(x, Int n, e2) -> e2{((Int n)/x}
  | Let(x, e1, e2) -> Let (x, step e1, e2)
**Substitution**

(* [subst e v x] is e{v/x}, that is, *
* [e] with [v] substituted for [x]. *)

```ml
let rec subst e v x = match e with
  | Var y -> if x=y then v else e
  | Int n -> Int n
  | Add(el,er) ->
    Add(subst el v x, subst er v x x)
  | Let(y,ebind,ebody) ->
    let ebind' = subst ebind v x in
    if x=y
    then Let(y, ebind', ebody)
    else Let(y, ebind', subst ebody v x)
```
let rec step = function

| Int n -> failwith "Does not step"
| Add(Int n1, Int n2) -> Int (n1 + n2)
| Add(Int n1, e2) -> Add (Int n1, step e2)
| Add(e1, e2) -> Add (step e1, e2)
| Var _ -> failwith "Unbound variable"
| Let(x, Int n, e2) -> subst e2 (Int n) x
| Let(x, e1, e2) -> Let (x, step e1, e2)
Upcoming events

• [Wednesday] A3 due