Assignment 3: Lambda Interpreter

Due: Wednesday, October 18, 2017 at 4:20pm
Submission cutoff: Saturday, October 21, 2017 at 4:20pm

In this assignment, you will implement an interpreter for a variant of the *simply typed lambda calculus* or $\lambda_{\rightarrow}$ as presented in class. This will help you become more familiar with the OCaml programming language as well as the process of writing functional interpreters for small languages, a common feature of language experimentation.

Setup

OCaml guide: OCaml setup

After setting up OCaml on your environment, copy the assignment folder.

On the Rice machines:

```
cp -r /afs/ir/class/cs242/assignments/assign3 assign3
```

On your local machine:

```
scp -r <SUNetID>@myth.stanford.edu:/afs/ir/class/cs242/assignments/assign3 assign3
```

Requirements

You must implement the typechecking and interpretation stages of the interpreter. You will turn in these files:

- `ast.ml`
- `typecheck.ml`
- `interpreter.ml`

You will also solve written problems in Part 4. You will submit a file `solutions.pdf` with your solutions.

Submitting

Then upload the files in your copy of `assign3` to Rice. To submit, navigate to the assignment root and execute:

```
python /afs/ir/class/cs242/bin/submit.py 3
```

Note: We are grading tests that do not result in an error using exact text matching, and will not fix submissions that have extra print statements or the like. It is your responsibility to test before submitting

In order to verify that your assignment was submitted, execute

```
ls /afs/ir/class/cs242/submissions/assign3/$USER
```

You should see the timestamps of your submissions.

Prelude: Modules

To understand the starter code, you will need to have a passing familiarity with one last OCaml concept: modules. For our purposes, modules provide basically just a namespace, or a way to separate out one function from another. Here's

an example:

```
module Counter = struct
  type t = int
  let make () : t = 0
  let incr (counter : t) : t = counter + 1
  let value (counter : t) : int = counter
end

let ctr : Counter.t = Counter.make ()
let ctr : Counter.t = Counter.incr ctr
print_int (Counter.value ctr)
```

A common convention in OCaml is that a module has a type `t` that represents the type associated with the module. For example, in our starter code, we have a `Term.t` that represents what a term is in the code, and a `Type.t` for types.

In an OCaml project, each file implicitly defines a new module, where the module's name is the same as the file's. For example, if I defined a file `foo.ml` that contained:

```
let do_something () = print_string "Hello"
```

Then from a file `bar.ml` in the same directory, I can run:

```
let () = Foo.do_something ()
```

You don't need requires/imports or anything, that happens implicitly. Lastly, within our source directory, for each `.ml` file we have a corresponding `.mli` file, where the `i` means "interface." These define the type signature of the function contained in the file (just like a `.h` file in C). You do not need to modify these files. We will talk more about modules in class next week.

Language specification

You will be implementing $\lambda_{\to}$, the simply typed lambda calculus (with a few twists). The language syntax is shown below. The first two columns are abstract and concrete syntax, and the third column is the English name.

Grammar

Type τ ::=	$\texttt{Int}$	$\texttt{int}$	integer
	$\texttt{Fn}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$	function
Term t ::=	$\texttt{Var}(x)$	x	variable
	$\texttt{Int}(n)$	n	integer
	$\texttt{Binop}(\oplus, t_1, t_2)$	$t_1 \oplus t_2$	binary operation
	$\texttt{Lam}(x, \tau, t')$	$\texttt{fn}\ (x : \tau)\ .\ t'$	function
	$\texttt{App}(t_1, t_2)$	$t_1\ t_2$	application
Binop $\oplus$ ::=	$\texttt{Add}$	$+$	addition
	$\texttt{Sub}$	$-$	subtraction
	$\texttt{Mul}$	$*$	multiplication
	$\texttt{Div}$	$/$	division

Statics

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(T-Var)} \qquad \frac{}{\texttt{Int}(n) : \texttt{int}}\ \text{(T-Int)} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\texttt{fn}\ (x : \tau_1)\ .\ t) : \tau_1 \to \tau_2}\ \text{(T-Lam)}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}\ \text{(T-App)} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \quad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 \oplus t_2 : \texttt{int}}\ \text{(T-Binop)}$$

Dynamics

$$\frac{}{\texttt{Int}(n)\ \text{val}}\ \text{(D-Int)} \qquad \frac{}{(\texttt{fn}\ (x : \tau)\ .\ t)\ \text{val}}\ \text{(D-Lam)}$$

$$\frac{t_1 \mapsto t_1'}{(t_1\ t_2) \mapsto (t_1'\ t_2)}\ \text{(D-App}_1\text{)} \qquad \frac{t_1\ \text{val} \quad t_2 \mapsto t_2'}{(t_1\ t_2) \mapsto (t_1\ t_2')}\ \text{(D-App}_2\text{)} \qquad \frac{t_2\ \text{val}}{((\texttt{fn}\ (x : \tau)\ .\ t_1)\ t_2) \mapsto [x \to t_2]\ t_1}\ \text{(D-App}_3\text{)}$$

$$\frac{t_1 \mapsto t_1'}{t_1 \oplus t_2 \mapsto t_1' \oplus t_2}\ \text{(D-Binop}_1\text{)} \qquad \frac{t_1\ \text{val} \quad t_2 \mapsto t_2'}{t_1 \oplus t_2 \mapsto t_1 \oplus t_2'}\ \text{(D-Binop}_2\text{)}$$

$$\frac{}{t_1\ /\ \texttt{Int}(0)\ \text{error}}\ \text{(D-Div)} \qquad \frac{}{\texttt{Int}(n_1) \oplus \texttt{Int}(n_2) \mapsto \texttt{Int}(n_1 \oplus n_2)}\ \text{(D-Binop}_3\text{)}$$

Code overview

Before you start writing the interpreter, read through the code for the interpreter using the arithmetic language defined in class: link. It will help you understand many of the concepts in this assignment.

Now, take a moment to familiarize yourself with all of the starter code provided. The flow of the interpreter is:

1. Raw source code enters as a string at the toplevel `main.ml` .
2. The source string is tokenized (lexed) into tokens in `lexer.mll` .
3. The tokens are parsed in `parser.mly` into an abstract syntax tree (AST) defined in `ast.mli` .
4. The AST is typechecked in `typecheck.ml` .
5. The AST is interpreted to produce a final result in `interpreter.ml` .

Each set of `.ml` and `.mli` files represent a module, with the `.mli` as the *interface* and `.ml` as the *implementation*. For example, `typecheck.mli` provides the interface to the typechecker, whereas `main.ml` has no corresponding interface as no other functions call the main.

Part 1: Substitution

An essential part of the lambda calculus is properly dealing with scope, binding and shadowing. This will be primarily addressed through your implementation of variable substitution. Specifically, you will implement the following function in `ast.ml` :

```
val substitute : Var.t -> Term.t -> Term.t -> Term.t
```

This is approximate translation of $[x \to t']\ t =$ `substitute x t' t` . The `substitute` function substitutes every non-shadowed instance of `x` (i.e. every `x` that is a free variable) in `t` with `t'` . Here are a few examples of substitutions:

$$[x \to 2]\ (1 + x) = 1 + 2$$
$$[x \to 2]\ ((\texttt{fn}\ (x : \texttt{int})\ .\ x)\ x) = (\texttt{fn}\ (x : \texttt{int})\ .\ x)\ 2$$
$$[x \to 2]\ ((\texttt{fn}\ (y : \texttt{int})\ .\ y + x)\ x) = (\texttt{fn}\ (y : \texttt{int})\ .\ y + 2)\ 2$$

Part 2: Typechecker

The next step is to validate that a given term is well-typed according to the statics provided above. Your goal is to implement the `typecheck` function in `typechecker.ml` . It adheres to the following signature:

```
val typecheck : Term.t -> (Type.t, string) Result.t
```

The `typecheck` function takes in a Term and returns a `Result.t` , a sum type that is either `Ok(the_type)` if the term typechecks or `Error(the_error)` if it does not.

Again, your implementation of `typecheck` should mirror exactly the semantics described by the typing rules in the language specification. You will need to carry around a typing context, a mapping from variables to types. This can be concretely implemented using a `String.Map.t`, described further in `code_examples.ml`.

Here's a few examples:

```
typecheck(Term.Int(3)) = Ok(Type.Int)  (* 3 : int *)
typecheck(Term.App(Term.Int(3), Term.Int(3))) = Error("something informative")
typecheck(Term.Lam("x", Type.Int, Term.Int(3))) = Ok(Type.Fn(Type.Int, Type.Int))  (* (fn (x : int) . 3) : int -> in
```

Part 3: Interpreter

Once you have verified that a term is well-typed, the last step is to write an interpreter in `interpreter.ml` that attempts to reduce the term to a value. Your interpreter operates on *small-step semantics* as is given in the language specification–that is, your core interpreter routine will reduce the term one baby step at a time. Specifically, you will implement `trystep`:

```
type outcome =
  | Step of Term.t
  | Val
  | Err of string

val trystep : Term.t -> outcome
```

The `trystep` function attempts to take a single step on a well-typed term `t` strictly following the dynamics above. If `t` is value then the outcome is `Val`. If `t` successfully stepped, then the outcome is `Step t'` where $t \mapsto t'$. If `t` is in an error state (a divide by zero in this language), then an `Err` outcome should be returned.

Here's a few examples:

```
trystep(Term.Int(3)) = Val  (* 3 val *)

let lam = Term.Lam("x", Type.Int, Term.Int(3)) in
trystep(Term.App(lam, Term.Int(2))) = Step(Term.Int(3))  (* (fn (x : int) . 3) 2 |-> 3 *)
trystep(Term.Binop(Term.Add, Term.Binop(Term.Add, Term.Int(1), Term.Int(3)), Term.Int(7)))
        = Step(Term.Binop(Term.Add, Term.Int(4), Term.Int(7)))
        (* (1 + 3) + 7 |-> 4 + 7 *)
trystep(Term.Binop(Term.Div, Term.Int(3), Term.Int(0))) = Err("div by 0")
```

Part 4: Language extensions

This is the written portion of the assignment. Always eager to Move Fast and Break Things™, Will has submitted two new potential extensions to the language above. For each extension, he wrote down the proposed statics and dynamics. Unfortunately, only one of these extensions is type safe–the other violates at least one of progress and preservation. Your task is to identify which extension violates these theorems and provide a counterexample, then for the other extension provide a proof of both progress and preservation for the given rules.

First, recall from lecture the definitions of progress and preservation.

Progress: if $t : \tau$, then either t val or there exists an t' such that $t \mapsto t'$.

Preservation: if $t : \tau$ and $t \mapsto t'$ then $t' : \tau$.

The first extension adds `let` statements to the language, just like in OCaml:

$$\frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x : \tau_1 = t_1\ \texttt{in}\ t_2) : \tau_2}\ \text{(T-Let)} \qquad \frac{}{\texttt{let}\ x : \tau = t_1\ \texttt{in}\ t_2 \mapsto [x \to t_1]\, t_2}\ \text{(D-Let)}$$

For example, this would allow us to write:

$$\texttt{let}\ x : \texttt{int} = 3\ \texttt{in}\ x + 2$$

The second extension adds a recursor (`rec`) to the language. A recursor is like a for loop (or more accurately a "fold") that runs an expression from 0 to n:

$$\frac{\Gamma \vdash t : \texttt{int} \quad \Gamma \vdash t_0 : \tau \quad \Gamma, x : \texttt{int}, y : \tau \vdash t_1 : \tau}{\Gamma \vdash \texttt{rec}(t_0; x.\, y.\, t_1)(t) : \tau} \text{ (T-Rec)}$$

$$\frac{t \mapsto t'}{\texttt{rec}(t_0; x.\, y.\, t_1)(t) \mapsto \texttt{rec}(t_0; x.\, y.\, t_1)(t')} \text{ (D-Rec}_1\text{)} \qquad \frac{}{\texttt{rec}(t_0; x.\, y.\, t_1)(\texttt{Int}(0)) \mapsto t_0} \text{ (D-Rec}_2\text{)}$$

$$\frac{n \neq 0}{\texttt{rec}(t_0; x.\, y.\, t_1)(\texttt{Int}(n)) \mapsto [x \to \texttt{Int}(n), y \to \texttt{rec}(t_0; x.\, y.\, t_1)(\texttt{Int}(n-1))]\, t_1} \text{ (D-Rec}_3\text{)}$$

Here's a few examples of using a recursor:

$$\texttt{rec}(0; x.\, y.\, x + y)(n) = 0 + 1 + \ldots + n$$
$$\texttt{rec}(1; x.\, y.\, x * y)(n) = 1 * 1 * 2 * \ldots * n$$

If you know LaTeX, we strongly recommend that you use LaTeX to write your proofs. We have provided you a file `tex/solution.tex` in which you may write your solutions. You may, however, use a word processor or handwrite your solutions—we simply require that your submission be a PDF and legible.

Testing

To build the interpreter, run `make` . This will create an executable, `main.native` .

Running `run_tests.py` will run the test suite over all the files in the `tests` directory and compare them to our solution.

```
python3 run_tests.py
```

Note that the tests are just a representative sample of the tests we will run your code on.

To test an individual file, you can use `main.native` to invoke the interpreter manually, e.g.

```
./main.native tests/function.lam1
```

For each of the functions you have to implement, we have provided a few additional unit tests inside of an `inline_tests` function in each file. Uncomment the line that says:

```
let () = inline_tests ()
```

And then execute `./main.native` to run those tests. If it does not fail with an assertion error, you have passed those tests.

One last thing: we have provided you with a number of useful code examples in `code_examples.ml` that demonstrate libraries/features you will find useful on the assignment that we may not have covered in-depth in class. Read through that file before starting the assignment. It also generates a binary `code_examples.native` that you can modify and run.