CS440: Programming Languages and Translators

Lecture 11

Spring 2023
Environments with functions: first try

```ocaml
let add = fun (x, y) -> x + y
let three = add (1, 2)
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Environments with functions: first try

let add = fun (x, y) -> x + y
let three = add (1, 2)

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>fun (x, y) -&gt; x + y</td>
</tr>
</tbody>
</table>
Environments with functions: first try

let add = fun (x, y) -> x + y
let three = add (1, 2)

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>fun (x, y) -&gt; x + y</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>
Environments with functions: first try

let add = fun (x, y) -> x + y
let three = add (1, 2)

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>fun (x, y) -&gt; x + y</td>
</tr>
<tr>
<td>three</td>
<td>3</td>
</tr>
</tbody>
</table>
Environments with functions: first try

let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>fun x -&gt; fun y -&gt; x + y</td>
</tr>
</tbody>
</table>
Environments with functions: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```
Environments with functions: first try

let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>fun x -&gt; fun y -&gt; x + y</td>
</tr>
<tr>
<td>add1</td>
<td>fun y -&gt; x + y</td>
</tr>
</tbody>
</table>

Uh oh
Environments with functions: first try

\[
\begin{align*}
  \text{let } x &= 1 \text{ in} \\
  \text{let } f \ y &= x + y \text{ in} \\
  \text{let } x &= 2 \text{ in} \\
  f \ 2
\end{align*}
\]
Environments with functions: first try

let x = 1 in
let f y = x + y in
let x = 2 in
f 2

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>fun y -&gt; x + y</td>
</tr>
</tbody>
</table>
Environments with functions: first try

let x = 1 in
let f y = x + y in
let x = 2 in
f 2

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>fun y -&gt; x + y</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
</tbody>
</table>
Environments with functions: first try

let x = 1 in
let f y = x + y in
let x = 2 in
f 2

x should still be 1 in f!

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>fun y -&gt; x + y</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>
Second try: use closures

• Closure: function code + environment
• This will be the value of a function

• in ML:
  
  | VClos of var * expr * env and env = (var * value option ref) list
  
  type value = ...
With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```
With closures

let x = 1 in
let f y = x + y in
let x = 2 in
f 2
With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```
With closures

let x = 1 in
let f y = x + y in
let x = 2 in
f 2
With closures

let x = 1 in
let f y = x + y in
let x = 2 in
f 2

Call the function with the environment from the closure (+ arguments)
Interpreting with closures

• Interpreting a function: fun x -> e
  • Return a closure with variable x, expression e, current environment

• Interpreting an application e1 e2
  • Interpret e1 to closure (x, e, env)
  • Interpret e2 to arg value v
  • Add x -> v to env, interpret e with this env
Recursive closures

let n = 5 in
let rec f m =
  if m >= n then 1
  else m * f (m + 1)
in
f 0
Interpreting with recursive closures

- Interpreting a function def: let rec f x = e1 in e2
  - Let env' = current env extended with placeholder for f
  - Let clos = Vclos(x, e1, env')
  - Update env' with f -> clos
  - Evaluate e2 with env'

- Interpreting an application e1 e2
  - Same as before: when you evaluate e1 to a closure, the function is already in the environment