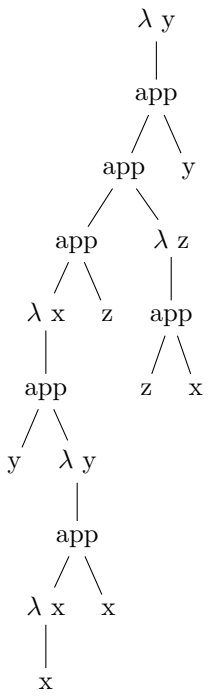
Programming Languages: Recitation 9

Goktug Saatcioglu

04.11.2019

1 HW7 Problem 1

1. The easiest way to solve this problem is to draw the AST of the expression. The AST is given below.



From the AST we see that the set of free variables are the first z and the second x. Alpha-renaming the tree gets us the expressions

$$t^{'} = \lambda\, y_1.\, (\lambda\, x_1.\, y_1\, (\lambda\, y_2\, .\, (\lambda\, x_2.\, x_2)\, x_1))\, z\, (\lambda\, z_1.\, z_1\, x)\, y_1$$

and the set of free variables are

$$\{x, z\}.$$

2. The step-by-step derivation of the result of the expression is given below.

$$
\begin{aligned}
\texttt{iszero}\ (\texttt{mult}\ \texttt{0}\ \texttt{1})\ \texttt{2}\ \texttt{3} &= \texttt{iszero}\ ((\lambda\, m\, n.\, m\ (\texttt{plus}\ n)\ \texttt{0})\ \texttt{0}\ \texttt{1})\ \texttt{2}\ \texttt{3} && \text{def. of } \texttt{mul} \\
&\xrightarrow{\beta} \texttt{iszero}\ ((\lambda\, n.\, \texttt{0}\ (\texttt{plus}\ n)\ \texttt{0})\ \texttt{1})\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{iszero}\ (\texttt{0}\ (\texttt{plus}\ \texttt{1})\ \texttt{0})\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&= \texttt{iszero}\ ((\lambda\, s\, z.\, z)\ (\texttt{plus}\ \texttt{1})\ \texttt{0})\ \texttt{2}\ \texttt{3} && \text{def. of } \texttt{zero} \\
&\xrightarrow{\beta} \texttt{iszero}\ ((\lambda\, z.\, z)\ \texttt{0})\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{iszero}\ \texttt{0}\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&= (\lambda\, n.\, n\ (\lambda\, x.\, \texttt{false})\ \texttt{true})\ \texttt{0}\ \texttt{2}\ \texttt{3} && \text{def. of } \texttt{iszero} \\
&\xrightarrow{\beta} \texttt{0}\ (\lambda\, x.\, \texttt{false})\ \texttt{true}\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&= (\lambda\, s\, z.\, z)\ (\lambda\, x.\, \texttt{false})\ \texttt{true}\ \texttt{2}\ \texttt{3} && \text{def. of } \texttt{zero} \\
&\xrightarrow{\beta} (\lambda\, z.\, z)\ \texttt{true}\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{true}\ \texttt{2}\ \texttt{3} && \text{reduce} \\
&= (\lambda\, x\, y.\, x)\ \texttt{2}\ \texttt{3} && \text{def. of } \texttt{true} \\
&\xrightarrow{\beta} (\lambda\, y.\, \texttt{2})\ \texttt{3} && \text{reduce} \\
&\xrightarrow{\beta} \texttt{2} && \text{reduce} \\
\therefore\ \texttt{iszero}\ (\texttt{mult}\ \texttt{0}\ \texttt{1})\ \texttt{2}\ \texttt{3} &= \texttt{2}
\end{aligned}
$$

3. Any number m^n can be thought as $m \times m \times m \times \cdots \times m$ where we multiply m n times. The Church encoding for a number n is given as

$$\lambda\, s\, z.\, s\ (s\ (\ldots\ (s\ z)))$$

which means that the function s is applied n times. Since we want to multiply n times the first argument of n should be (`mult m`). For z the only correct argument is `1` as in the base case we would like $m^0 = 1$ and if m^n where $n > 1$ we want to get the correct result. Bringing everything together, we get the following solution

$$\texttt{exp}\ =\ (\lambda\, m\, n.\, n\ (\texttt{mul}\ \texttt{m})\ \texttt{1}).$$

2 HW7 Problem 2

1. We skip the pretty printer solution for the sake of brevity but the principles for answering the questions below also apply here. The idea is to traverse the AST and use the OCaml printy printing to reconstruct the code. There are a few issues to pay attention to like the precedence of the operators for binary operations. See the sample solution for the detailed solution and feel free to contact me if anything is still unclear.

2. We are given term and we wish to find the first free variable. We know that the terms are defined by the algebraic data type `type term = ...` meaning we must account for all cases. We know that if we see a `FunConst`, `IntConst` or `BoolConst` there are no free variables as these terms are simply constants. If we see a a function application `App` then we need to check first the function term and then the argument term and if neither have free terms then `App` has no free term. Similarly, this idea applies to binary operators `BinOp` and if-then-else `Ite` with the latter being extended to checking whether the condition has a free variable, if not then checking the true case and if not then checking the false case. So we are only left with the cases `Lambda` and `Var`. We know that `Lambda` declares a variable such that all sub-occurences are bound to that variable. So if we have a `Lambda` and then some sub-terms that refer to the declared variable then there are no free terms. We need some way of holding onto all bound variables and we'll use a list for this. For each `Lambda` we update the bound

variables list and then traverse the body using the logic above. If we ever encounter a `Var` then we can check if it is bound by doing `List.mem x bound` where if the call returns `true` then we know this variable isn't free and otherwise it must be free. So we have a general idea for how to achieve the needed implementation. However, rather than `true` or `false` we need to return `None` and `Some x` at the point `Var`. How can we propagate the result of one sub-check such as done in `Ite` such that if one component returns `Some x` then we retain that one and if one component returns `None` and the next returns `Some x` we update our result? Furthermore, if all the answers are `None` then we should return `None`. The function `lazy_or_else` in `opt.ml` achieves our need of correctly propagating each check into the next check so we use it in our answer. So bringing everything together, we have a list of bound variables `bound` that we update on each `Lambda`, we check all sub-components of terms that are not base cases for free variables, if we get to a base case other than `Var` we return `None` and if the base case is `Var` then we check whether the variable is located in `bound` and return an answer accoringly. The solution is given below.

```
let find_free_var (t: term) : (var * pos) option =
  let rec fv bound = function
    | FunConst _ | IntConst _ | BoolConst _ -> None
    | Var (x, pos) ->
        if List.mem x bound
        then None
        else Some (x, pos)
    | App (t1, t2, _)
    | BinOp (_, t1, t2, _) ->
        fv bound t1 |>
        Opt.lazy_or_else (fv bound) t2
    | Ite (t1, t2, t3, _) ->
        fv bound t1 |>
        Opt.lazy_or_else (fv bound) t2 |>
        Opt.lazy_or_else (fv bound) t3
    | Lambda (x, t, _) ->
        fv (x :: bound) t
  in
  fv [] t
```

As an extra note, using a list is slightly inefficient as all calls to `List.mem` will traverse the whole list. How can we make this implementation more efficient? Consider the OCaml module `Set` where its function calls are specified here. We learned about functors in yesterday's class and if you would like to try implementing the `Set` functor consider the example given here which shows how to create a `Set` of type `Int`. We would like to have a `Set` of type `var` and pass this set named `bound` accordingly.

3. The idea to this question is similar to what we did for the previous question. We must traverse the AST and consider all cases where there are variables and replace them if they are the ones we are interested in. From the lambda calculus lecture we know that all the bound occurences of `s` in λ `s. t` are the free occurrences of `s` in `t`. So we have the term `t` and everytime we see an `x` we will update it with some new term `s`. If we see `Var` that is equivalent to `x` then we return `s` to perform the update. If see an `App`, `BinOp` or `Ite` we will traverse all sub-terms to find the free-occurrences there and update them if necessary. If we see a `Lambda` then we should only consider its body if and only if its argument is not equal to what we wish to change. Why is this the case? Consider the term λ `s.` (λ `s. t`) `t`. The references to `s` in the inner are bound to the inner decaration λ `s. t` while the references to `s` in the outer `t` refer to the outer declaration of `s`. So if we are given the above term then we must only update the free occurences of `s` in the outer `t` and ignore those in the inner λ `s. t`. Finally, since we have identified all the cases where an update is necessary if we come across any other term

it is sufficient to simply return it. For example, if we see an `IntConst (i,pos)` we should return the same `IntConst (i,pos)`. The solution to the problem is given below.

```
let subst (t: term) (x: var) (s: term) =
  let rec st = function
    | Var (y, _) when x = y -> s
    | App (t1, t2, pos) -> App (st t1, st t2, pos)
    | BinOp (bop, t1, t2, pos) -> BinOp (bop, st t1, st t2, pos)
    | Ite (t1, t2, t3, pos) -> Ite (st t1, st t2, st t3, pos)
    | Lambda (y, t1, pos) when x <> y -> Lambda (y, st t1, pos)
    | t -> t
  in st t
```

4. To answer this question we need to look at how the function `beta_call_by_value` works. The first line evaluates the first term meaning we evaluate the function body first. The first term must be a lambda as this is function application so we get a closure with its first argument being its parameter and the second argument being the function body. The second line retrieves the contents of the closure and if the first line returned anything other than a closure we would get an error as then we wouldn't have a valid function application. The third line evaluates the second term meaning we evaluate the function argument and now this doesn't have to be a lambda. With `(subst t x (term_of_value (position_of_term t2) v2))` we substitute all free occurences of the argument of the created closure with the evaluated argument. This is the beta-reduction step that gives us call by name semantics. Then we take this result and continue to evaluate by doing `eval beta_call_by_value`. So for a call by name implementation we can simply modify this implementation. We should keep the first two lines the same as we need to evaluate the first term and get the contents of the closure. Now as we have call by name semantics we shouldn't evaluate the second term and instead do the substitution directly so as to get textual-replacement of the relevant terms. The implementation is given below.

```
let rec beta_call_by_name  (t1: term) (t2: term) (pos: pos) : value =
  let v1 = eval beta_call_by_name t1 in
  let x, t, _ = closure_of_value (position_of_term t1) v1 in
  eval beta_call_by_name (subst t x t2)
```

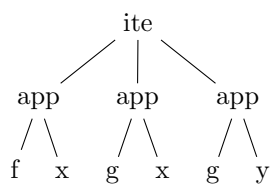
5. We will also not cover the bonus question during the recitation. The ideas for solving the problem are esentially the same as those used for solving parts 1 through 4. However, this time we need to pass around an environment and evaluate the expressions using this environment. The environment is basically an empty list. Again, feel free to contact me if you have any questions about the sample solution or would like to talk about your implementation.

3 Type Inference

- In Class 09 we learned about type systems and type inference. Let us try inferring the type of the following expression:

  `let h f g x y = if (f x) then (g x) else (g y).`

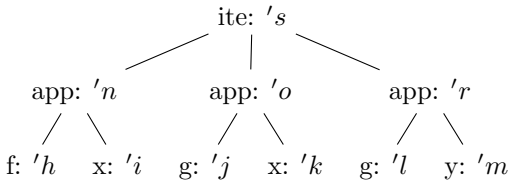
  We begin by drawing the AST of the body of the function. This is given below.



Then we give type names to the function arguments and the result. So our function h is of type

$$h : {'f} \to {'g} \to {'x} \to {'y} \to {'s}$$

where $'s$ is the type the function returns. Next, we label each node of the AST with a type name so that each subexpression gets a type. This looks as follows.



Now we can proceed to create our constraints. We begin with the simple ones first and this involves simply looking at the leaves of the AST.

$$\begin{aligned} {'f} &= {'h} \\ {'x} &= {'i} \\ {'g} &= {'j} \\ {'x} &= {'k} \\ {'g} &= {'l} \\ {'y} &= {'m} \end{aligned}$$

Now let's generate the constraints involed with the functions applications. For example the first application returns a type $'n$ so $'f$ must be a function that takes a type $'i$ and return a type $'n$. Similar reasoning applies to the other application nodes.

$$\begin{aligned} {'f} &= {'i} \to {'n} \\ {'j} &= {'k} \to {'o} \\ {'l} &= {'m} \to {'r} \end{aligned}$$

Then, we have to consider the context of the if-then-else sub-expression. We know that the condition must return a type of `bool` and the two sub-expressions in the branches must return the same type which in this case is $'s$ as this is what the function returns. So we proceed to get three more constraints which are given below.

$$\begin{aligned} {'n} &= bool \\ {'o} &= {'s} \\ {'r} &= {'s} \end{aligned}$$

Now we proceed to solve the constraints to find the most general unifier. Let's proceed from the last constraints which gives us the most general unifier

$$\sigma_1 = \{{'n} \mapsto bool, {'o} \mapsto {'s}, {'r} \mapsto {'s}\}.$$

We apply σ_1 to ${'l} = {'m} \to {'r}$ getting us

$$\sigma_2 = \{{'n} \mapsto bool, {'o} \mapsto {'s}, {'r} \mapsto {'s}, {'l} \mapsto {'m} \to {'s}\}$$

which we then apply to ${'j} = {'k} \to {'o}$ getting us

$$\sigma_3 = \{{'n} \mapsto bool, {'o} \mapsto {'s}, {'r} \mapsto {'s}, {'l} \mapsto {'m} \to {'s}, {'j} \mapsto {'k} \to {'s}\}$$

which we then apply to $'f = 'i \to 'n$ getting us

$$\sigma_4 = \{'n \mapsto bool, 'o \mapsto 's, 'r \mapsto 's, 'l \mapsto 'm \to 's, 'j \mapsto 'k \to 's, 'f \mapsto 'i \to bool\}.$$

We then apply σ_4 to the constraints we derived from the leaves of the AST. We skip over the step-by-step derivation of these constraints as they are pretty straightforward. We can simply replace each occurrence of a type in σ_4 with the corresponding introduced constraint and then add the constraint to the new σ_{k+1}. The most general unifier after 6 more steps is given below.

$$\begin{aligned}\sigma_{10} = \{&'n \mapsto bool, 'o \mapsto 's, 'r \mapsto 's, 'g \mapsto 'x \to 's, 'h \mapsto 'x \to bool \\ &'m \mapsto 'y, 'k \mapsto 'x, 'i \mapsto 'x, 'j \mapsto 'g, 'l \mapsto 'g, 'f \mapsto 'h, 'y \mapsto 'x\}\end{aligned}$$

We apply this unifier to the original function signature

$$h : 'f \to 'g \to 'x \to 'y \to 's$$

which gives us the final answer

$$h : ('x \to bool) \to ('x \to 's) \to 'x \to 'x \to 's.$$

Note that we can rename all the type variables except the conrete ones to get what putting this expression would give us. So we get the following equivalent typing in OCaml

$$h : ('a \to bool) \to ('a \to 'b) \to 'a \to 'a \to 'b.$$

4 Type Inhabitation

- We just took a look at the type inference problem. The inverse problem is know as the type inhabitation problem. That is, given a type, the goal is to infer an expression that has that type.
- We know that there are two approaches to solving this problem. The first is to systematically analyze the structure of the type expression and get the answer. The second it to rely on the Curry-Howard correspondence. Roughly speaking, the Curry-Howard Correspondence establishes an equivalence between inferring an expression for a given type and the problem of proving the validity of a logical formula obtained from that type.
- For more information on the Curry-Howard Correspondence you can start here.
- Consider the following ADT.

```
type ('a, 'b) either =
| Left of 'a
| Right of 'b
```

- We take the Curry-Howard approach first. The polymorphic types we are dealing with here can be interpreted as follows.
  - Type variables like `'a` and `'b` correspond to proposition variables that take on the truth values true and false.
  - Function types `'a -> 'b` correspond to logical implications (written `'a => 'b`). Here, `'a => 'b` states that if `'a` is true then so is `'b`.
  - Product types `'a * 'b` correspond to logical conjunctions (written `'a && 'b`). Here `'a && 'b` is true iff both `'a` and `'b` is true.

- Either types `('a, 'b)` either correspond to logical disjunctions (written `'a || 'b`). Here `'a || 'b` is true iff one of `'a` of `'b` is true.

- We wish to find a program that satisfies the following type expressions:

  ```
  (('x,'y) either, 'z) either -> ('x, ('y,'z) either) either
  ```

  which using the logical formulas from the above gets us

  ```
  (x || y) || z = x || (y || z).
  ```

- Now we proceed with case analysis on the formula to obtain the probem. We know the whole expression is `true` so we can start with either `z` is `true` of `(x || y)` is `true`.

  - Case 1. If `z` is `true` then `(y || z)` is `true` so that `x || (y || z)` is `true`. So to consider `z` we must match on `Right (z)`. Then the statement `(y || z)` is `true` is equivalent to `Right (Right (z))`. This completes the first case.
  - Case 2. The statement `(x || y)` is `true`.
    * Case 2.1. If `y` is `true` then `(y || z)` is `true` so that `x || (y || z)` is `true`. So to consider `y` is `true` we must match on `Left ( Right (z))`. Then the statement `(y || z)` is `true` is equivalent to `Right ( Left (y))`.
    * Case 2.2. If `x` is `true` then `x || (y || z)` is `true`. So to consider `x` is `true` we must match on `Left ( Left (z))`. Then the statement `x || (y || z)` is `true` is equivalent to `Left (x)`.

  Bringing everything together from our above analysis, we see that the function we want is

  ```
  let f e = match e with
  | Right (z) -> Right ( Right (z))
  | Left ( Right (y)) -> Right ( Left (y))
  | Left ( Left (x)) -> Left (x)
  ```

- We can also take the approach of analyzing the structure of the type expression. Recall that the type expression was

  ```
  (('x,'y) either, 'z) either -> ('x, ('y,'z) either) either
  ```

  and the ADT is

  ```
  type ('a, 'b) either =
  | Left of 'a
  | Right of 'b
  ```

  so that `(('x,'y) either, 'z) either` is pseudo-equivalent to `Left ( Left (x), Right (y)), Right (z)`. Note that "pseudo-equivalent" is a made-up term I created. The idea is that we have an inner either of type `'x` or type `'y` so expression of the form `Left (x)` and `Left (y)` match this sttructure. Then we have another either that has the inner either as its type in its left statement so we get `Left ( Left (x))` and `Left (Right (y))` to get the left type right. Since on the right we simply have type `'z` then this is `Right (z)`. So we now know what our pattern matching looks like. Let's write a part of the program we wish to obtain down.

```
let f e = match e with
| Right (z) -> ???
| Left ( Right (y)) -> ???
| Left ( Left (x)) -> ???
```

We wish to find the entries to the `???`'s. Now let's look at the result type which is

`('x, ('y,'z) either) either`.

So we consider the outer either where on the left we have of type `'x` which is of the form `Left (x)`. This should go in the place of the last `???`. The right of the outer either is another either so we will have the form of `Right (...)`. Since the type `'y` is on the left of the inner either we get the form `Right ( Left (y))` since the inner either is on the right of the outer either. This should go in the place of the middle `???`. Finally, the type `'z` is on the right of the inner either we get the form `Right ( Right (z))` since the inner either is on the right of the outer either. This should go in the place of the first `???`. Bringing everything together we get

```
let f e = match e with
| Right (z) -> Right ( Right (z))
| Left ( Right (y)) -> Right ( Left (y))
| Left ( Left (x)) -> Left (x)
```

which is the same answer as the solution we got from the Curry-Howard correspondence. (And it should be as this is what it states!)

- As a final note, observe how the structural apporach requires us to enumerate all possibilities of the structure of possible structures of the type expression. By doing this we get what we should match on and what we should return.