COMP3161/COMP9164
Syntax Exercises

Liam O'Connor

September 26, 2019

1. (a) [$\star$] Consider the following expressions in Higher Order abstract syntax. Convert them to concrete syntax.

   i. $(\texttt{Let}\ (\texttt{Num}\ 3)\ (x.\ (\texttt{Let}\ (\texttt{Plus}\ x\ (\texttt{Num}\ 1))\ (x.\ (\texttt{Plus}\ x\ x)))))$

   > **Solution:** $\textbf{let}\ x = 3\ \textbf{in}\ \textbf{let}\ x = x + 1\ \textbf{in}\ x + x$

   ii. $(\texttt{Plus}\ (\texttt{Let}\ (\texttt{Num}\ 3)\ (x.\ (\texttt{Plus}\ x\ x)))\ (\texttt{Let}\ (\texttt{Num}\ 2)\ (y.\ (\texttt{Plus}\ y\ (\texttt{Num}\ 4)))))$

   > **Solution:** $(\textbf{let}\ x = 3\ \textbf{in}\ x + x) + (\textbf{let}\ y = 2\ \textbf{in}\ y + 4)$

   iii. $(\texttt{Let}\ (\texttt{Num}\ 2)\ (x.\ (\texttt{Let}\ (\texttt{Num}\ 1)\ (y.\ (\texttt{Plus}\ x\ y)))))$

   > **Solution:** $\textbf{let}\ x = 2 + (\textbf{let}\ y = 1\ \textbf{in}\ x + y)$

   (b) [$\star$] Apply the substitution $x := (\texttt{Plus}\ z\ 1)$ to the following expressions:

   i. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (y.\ (\texttt{Plus}\ x\ y)))$

   > **Solution:** $(\texttt{Let}\ (\texttt{Plus}\ (\texttt{Plus}\ z\ 1)\ z)\ (y.\ (\texttt{Plus}\ (\texttt{Plus}\ z\ 1)\ y)))$

   ii. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (x.\ (\texttt{Plus}\ z\ z)))$

   > **Solution:** $(\texttt{Let}\ (\texttt{Plus}\ (\texttt{Plus}\ z\ 1)\ z)\ (x.\ (\texttt{Plus}\ z\ z)))$

   iii. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (z.\ (\texttt{Plus}\ x\ z)))$

   > **Solution:** Undefined without applying α-renaming first. Can safely substitute after renaming the bound z to a: $(\texttt{Let}\ (\texttt{Plus}\ (\texttt{Plus}\ z\ 1)\ z)\ (a.\ (\texttt{Plus}\ (\texttt{Plus}\ z\ 1)\ a)))$

   (c) [$\star$] Which variables are shadowed in the following expression and where?

$$(\texttt{Let}\ (\texttt{Plus}\ y\ 1)\ (x.\ (\texttt{Let}\ (\texttt{Plus}\ x\ 1)\ (y.\ (\texttt{Let}\ (\texttt{Plus}\ x\ y)\ (x.\ (\texttt{Plus}\ x\ y)))))))$$

   > **Solution:** The innermost let shadows the binding of x from the outermost let. The middle let shadows the free y mentioned in the outermost let.

2. Here is a concrete syntax for specifying binary logic gates with convenient $\texttt{if} - \texttt{then} - \texttt{else}$ syntax. Note that the `else` clause is optional, which means we must be careful to avoid ambiguity – we introduce mandatory parentheses around nested conditionals:

$$\frac{}{\top\ \textsc{Output}} \quad \frac{}{\bot\ \textsc{Output}} \quad \frac{}{\alpha\ \textsc{Input}} \quad \frac{}{\beta\ \textsc{Input}}$$

$$\frac{c\ \textsc{Input} \quad t\ \textsc{IExpr} \quad e\ \textsc{Expr}}{\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e\ \textsc{Expr}} \quad \frac{c\ \textsc{Input} \quad t\ \textsc{IExpr}}{\texttt{if}\ c\ \texttt{then}\ t\ \textsc{Expr}} \quad \frac{x\ \textsc{Output}}{x\ \textsc{IExpr}}$$

$$\frac{e\ \textsc{Expr}}{(e)\ \textsc{IExpr}} \qquad \frac{e\ \textsc{IExpr}}{e\ \textsc{Expr}}$$

If an `else` clause is omitted, the result of the expression if the condition is false is defaulted to $\bot$. For example, an `AND` or `OR` gate could be specified like so:

$$\texttt{AND} : \texttt{if}\ \alpha\ \texttt{then}\ (\texttt{if}\ \beta\ \texttt{then}\ \top)$$

$$\texttt{OR} : \texttt{if}\ \alpha\ \texttt{then}\ \top\ \texttt{else}\ (\texttt{if}\ \beta\ \texttt{then}\ \top)$$

Or, a `NAND` gate:

$$\texttt{if}\ \alpha\ \texttt{then}\ (\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \texttt{else}\ \top$$

(a) [$\star\star$] Devise a suitable *abstract syntax* A for this language.

Solution:

$$\frac{x \in \{\mathsf{a}, \mathsf{b}\}}{x\ \textsc{Input}} \qquad \frac{x \in \{\mathtt{T}, \mathtt{F}\}}{x\ \textsc{Output}} \qquad \frac{c\ \textsc{Input} \quad t\ \mathrm{A} \quad e\ \mathrm{A}}{\mathtt{If}\ c\ t\ e\ \mathrm{A}} \qquad \frac{x\ \textsc{Output}}{x\ \mathrm{A}}$$

(b) [$\star$] Write rules for a *parsing relation* ($\longleftrightarrow$) for this language.

Solution:

$$\frac{}{\top\ \textsc{Output} \longleftrightarrow \mathtt{T}}\textsc{Top} \qquad \frac{}{\bot\ \textsc{Output} \longleftrightarrow \mathtt{F}}\textsc{Bot} \qquad \frac{}{\alpha\ \textsc{Input} \longleftrightarrow \mathtt{A}}\textsc{Input}_\alpha \qquad \frac{}{\beta\ \textsc{Input} \longleftrightarrow \mathtt{B}}\textsc{Input}_\beta$$

$$\frac{c\ \textsc{Input} \longleftrightarrow c' \quad t\ \textsc{IExpr} \longleftrightarrow t' \quad e\ \textsc{Expr} \longleftrightarrow e'}{\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e\ \textsc{Expr} \longleftrightarrow \mathtt{If}\ c'\ t'\ e'}\textsc{If}_1 \qquad \frac{c\ \textsc{Input} \longleftrightarrow c' \quad t\ \textsc{IExpr} \longleftrightarrow t'}{\texttt{if}\ c\ \texttt{then}\ t\ \textsc{Expr} \longleftrightarrow \mathtt{If}\ c'\ t'\ \mathtt{F}}\textsc{If}_2$$

$$\frac{e\ \textsc{Expr} \longleftrightarrow e'}{(e)\ \textsc{IExpr} \longleftrightarrow e'}\textsc{Paren} \qquad \frac{e\ \textsc{Output} \longleftrightarrow e'}{e\ \textsc{IExpr} \longleftrightarrow e'}\textsc{Shunt}_1 \qquad \frac{e\ \textsc{IExpr} \longleftrightarrow e'}{e\ \textsc{Expr} \longleftrightarrow e'}\textsc{Shunt}_2$$

(c) [$\star$] Here's the parse derivation tree for the `NAND` gate above:

$$\dfrac{\dfrac{}{\alpha\ \textsc{Input} \longleftrightarrow} \qquad \dfrac{\dfrac{\dfrac{}{\beta\ \textsc{Input} \longleftrightarrow} \quad \dfrac{\dfrac{}{\bot\ \textsc{Output} \longleftrightarrow}}{\bot\ \textsc{IExpr} \longleftrightarrow} \quad \dfrac{\dfrac{\dfrac{}{\top\ \textsc{Output} \longleftrightarrow}}{\top\ \textsc{IExpr} \longleftrightarrow}}{\top\ \textsc{Expr} \longleftrightarrow}}{\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top\ \textsc{Expr} \longleftrightarrow}}{(\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \textsc{IExpr} \longleftrightarrow} \qquad \dfrac{\dfrac{\dfrac{}{\top\ \textsc{Output} \longleftrightarrow}}{\top\ \textsc{IExpr} \longleftrightarrow}}{\top\ \textsc{Expr} \longleftrightarrow}}{\texttt{if}\ \alpha\ \texttt{then}\ (\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \texttt{else}\ \top\ \textsc{Expr} \longleftrightarrow}$$

Fill in the right-hand side of this derivation tree with your parsing relation, labelling each step as you progress down the tree.

Solution:

$$\dfrac{\dfrac{}{\alpha\ \textsc{Input} \longleftrightarrow \mathtt{A}} \qquad \dfrac{\dfrac{\dfrac{}{\beta\ \textsc{Input} \longleftrightarrow \mathtt{B}} \quad \dfrac{\dfrac{}{\bot\ \textsc{Output} \longleftrightarrow \mathtt{F}}}{\bot\ \textsc{IExpr} \longleftrightarrow \mathtt{F}} \quad \dfrac{\dfrac{\dfrac{}{\top\ \textsc{Output} \longleftrightarrow \mathtt{T}}}{\top\ \textsc{IExpr} \longleftrightarrow \mathtt{T}}}{\top\ \textsc{Expr} \longleftrightarrow \mathtt{T}}}{\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top\ \textsc{Expr} \longleftrightarrow \mathtt{If\ B\ F\ T}}}{(\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \textsc{IExpr} \longleftrightarrow \mathtt{If\ B\ F\ T}} \qquad \dfrac{\dfrac{\dfrac{}{\top\ \textsc{Output} \longleftrightarrow \mathtt{T}}}{\top\ \textsc{IExpr} \longleftrightarrow \mathtt{T}}}{\top\ \textsc{Expr} \longleftrightarrow \mathtt{T}}}{\texttt{if}\ \alpha\ \texttt{then}\ (\texttt{if}\ \beta\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \texttt{else}\ \top\ \textsc{Expr} \longleftrightarrow \mathtt{If\ A\ (If\ B\ F\ T)\ T}}$$

3. Here is a *first order abstract syntax* for a simple functional language, $\textsc{Lc}$. In this language, a `lambda` term defines a *function.* For example, `lambda` x (`var` x) is the identity function, which simply returns its input.

$$\frac{e_1\ \textsc{Lc} \quad e_2\ \textsc{Lc}}{\mathtt{App}\ e_1\ e_2\ \textsc{Lc}} \qquad \frac{x\ \textsc{VarName} \quad e\ \textsc{Lc}}{\mathtt{Lambda}\ x\ e\ \textsc{Lc}} \qquad \frac{x\ \textsc{VarName}}{\mathtt{Var}\ x\ \textsc{Lc}}$$

(a) [$\star$] Give an example of *name shadowing* using an expression in this language, and provide an α-*equivalent* expression which does not have shadowing.

> **Solution:** A simple example is `Lambda x (Lambda x (Var x))`. Here, the name `x` is shadowed in the inner binding.
>
> An α-equivalent expression without shadowing would use a different variable `y`, i.e
>
> $$\texttt{Lambda x (Lambda y (Var } y\texttt{))}$$

(b) [$\star\star$] Here is an incorrect substitution algorithm for this language:

$$\begin{array}{lcl}
(\texttt{App}\ e_1\ e_2)[v := t] & \mapsto & \texttt{App}\ (e_1[v := t])\ (e_2[v := t]) \\
(\texttt{Var}\ v)[v := t] & \mapsto & t \\
(\texttt{Lambda}\ x\ e)[v := t] & \mapsto & \texttt{Lambda}\ x\ (e[v := t])
\end{array}$$

What is wrong with this algorithm? How can you correct it?

> **Solution:** The substitution doesn't deal with name clashes. The rule for lambdas should look like this:
>
> $$(\texttt{Lambda}\ x\ e)[v := t] \quad \mapsto \quad \begin{cases} \texttt{Lambda}\ x\ (e[v := t]) & \text{if } x \neq v \text{ and } x \notin FV(t) \\ \texttt{Lambda}\ x\ e & \text{if } x = v \\ \text{undefined} & \text{otherwise} \end{cases}$$

(c) [$\star\star$] Aside from the difficulties with substitution, using arbitrary strings for variable names in first-order abstract syntax means that α-equivalent terms can be represented in many different ways, which is very inconvenient for analysis. For example, the following two terms are equivalent, but have different representations:

`Lambda x (Lambda y (App (Var x) (Var y)))`

`Lambda a (Lambda b (App (Var a) (Var b)))`

One technique to achieve *canonical* representations (i.e α-equivalence is the same as equality) is called *higher order abstract syntax* (HOAS). Explain what HOAS is and how it solves this problem.

> **Solution:** Higher order abstract syntax encodes abstraction in the *meta-logic* level, or in the *language implementation*, rather than as a first-order abstract syntax construct.
>
> First order abstract syntax might represent a term like $\lambda x.x$ as something like `Lambda` "x" (`Var` "x"), where literal *variable name strings* are placed in the abstract syntax directly.
>
> Higher order abstract syntax, however, would place a *function* inside the abstract syntax, i.e `Lambda` ($\lambda x.\ x$), where the variable x is a *meta-variable* (or a variable in the language used to implement our interpreter, rather than the language being implemented). This function is (extensionally) equal to any other α-equivalent function, and therefore we can consider two α-equivalent terms to be equal with HOAS, assuming extensionality (that is, a function f equals a function g if and only if, for all x, $f(x) = g(x)$.
>
> For example, a first order Haskell implementation of the above syntax might look like this:
>
> ```
> type VarName = String
> data AST = App AST AST
>          | Var VarName
>          | Lambda VarName AST
> test = Lambda "x" (Lambda "y" (App (Var "x") (Var "y")))
> ```
>
> Whereas a higher order syntax might look like this:
>
> ```
> data AST = App AST AST
>          | Lambda (AST -> AST)
> test = Lambda $ \x -> Lambda $ \y -> App x y
> ```

There is no way in Haskell, for example, to determine that we used the names `x` and `y` for those function arguments. The only way for a Haskell function `f` to be distinguished from a function `g` is for `f` x to be different from `g` x for some x (i.e extensionality). As α-equivalent Haskell functions cannot be so distinguished, we must judge a term as equal to any other in its α-equivalence class.