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. $(\texttt{Let}\ (\texttt{Num}\ 3)\ (x.\ (\texttt{Let}\ (\texttt{Plus}\ x\ (\texttt{Num}\ 1))\ (x.\ (\texttt{Plus}\ x\ x)))))$
   ii. $(\texttt{Plus}\ (\texttt{Let}\ (\texttt{Num}\ 3)\ (x.\ (\texttt{Plus}\ x\ x)))\ (\texttt{Let}\ (\texttt{Num}\ 2)\ (y.\ (\texttt{Plus}\ y\ (\texttt{Num}\ 4)))))$
   iii. $(\texttt{Let}\ (\texttt{Num}\ 2)\ (x.\ (\texttt{Let}\ (\texttt{Num}\ 1)\ (y.\ (\texttt{Plus}\ x\ y)))))$

   (b) [$\star$] Apply the substitution $x := (\texttt{Plus}\ z\ 1)$ to the following expressions:
   i. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (y.\ (\texttt{Plus}\ x\ y)))$
   ii. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (x.\ (\texttt{Plus}\ z\ z)))$
   iii. $(\texttt{Let}\ (\texttt{Plus}\ x\ z)\ (z.\ (\texttt{Plus}\ x\ z)))$

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

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

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

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

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

$$\dfrac{e\ \textsc{Expr}}{(e)\ \textsc{IExpr}} \quad \dfrac{e\ \textsc{IExpr}}{e\ \textsc{Expr}}$$

If an $\texttt{else}$ 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.

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

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

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

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

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.

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

(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.

(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?

(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.