COMP3161/COMP9161

# Syntax Exercises

Liam O'Connor-Davis

August 9, 2018

1. Here is a concrete syntax for specifying binary logic gates with convenient `if − then − 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:

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

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

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

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:

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

Or, a `NAND` gate:

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

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

**Solution:**

$$\frac{x \in \{\texttt{a}, \texttt{b}\}}{x \textsc{ Input}} \quad \frac{x \in \{\texttt{T}, \texttt{F}\}}{x \textsc{ Output}} \quad \frac{c \textsc{ Input} \quad t \text{ A} \quad e \text{ A}}{\texttt{If } c\ t\ e \text{ A}} \quad \frac{x \textsc{ Output}}{x \text{ A}}$$

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

**Solution:**

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

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

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

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

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

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

**Solution:**

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

2. Here is a *first order abstract syntax* for a simple functional language, 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) [⋆] Give an example of *name shadowing* using an expression in this language, and provide an $\alpha$-*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 $\alpha$-equivalent expression without shadowing would use a different variable `y`, i.e

`Lambda x (Lambda y (Var` $y$`))`

(b) [⋆⋆] 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) [⋆⋆⋆] Aside from the difficulties with substitution, using arbitrary strings for variable names in first-order abstract syntax means that $\alpha$-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 $\alpha$-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 $\alpha$-equivalent function, and therefore we can consider two $\alpha$-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 $\alpha$-equivalent Haskell functions cannot be so distinguished, we must judge a term as equal to any other in its $\alpha$-equivalence class.