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.

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

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

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

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

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.

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

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

(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?

(c) [⋆⋆⋆] 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.