Concepts of Programming Language Design

Syntax Exercises

Liam O'Connor-Davis
Gabriele Keller

November 18, 2025

1. 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:

$$\frac{}{\top\ \textbf{Output}} \qquad \frac{}{\bot\ \textbf{Output}} \qquad \frac{}{\alpha\ \textbf{Input}} \qquad \frac{}{\beta\ \textbf{Input}}$$

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

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

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) Devise a suitable abstract syntax A for this language.

(b) Write rules for a parsing relation ($\leftrightarrow$) for this language.

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

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

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.

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

   (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.

3. Consider the following two definitions of a simple expression language deeply embedded in Haskell.

   First order embedding:

```haskell
data FOExpr
  = FONum Int
  | FOVar String
  | FOPlus FOExpr FOExpr
  | FOTimes FOExpr FOExpr
  | FOLetBnd String FOExpr FOExpr
```

   Higher order embedding:

```haskell
data HOExpr
  = HONum Int
  | HOPlus HOExpr HOExpr
  | HOTimes HOExpr HOExpr
  | HOLetBnd HOExpr (HOExpr -> HOExpr)
```

   (a) Define a function of type

```haskell
foToHO :: FOExpr -> HOExpr
```

   which converts a first order expression into the corresponding higher-order expression.

   (b) It is also possible to write a function of type

```haskell
hoToFO :: HOExpr -> FOExpr
```

   such that for all first order terms `t` with:

```haskell
hoToFO (foToHO t) = t'
```

the term `t'` is α-equivalent to `t`.

Hint: it is necessary to extend the data type of the higher-order representation with one additional data constructor, which is used during the transformation.

(c) Not for all higher order terms `h` it will hold that

```
foToHO  (hoToFO h) = h
```

Give a counter example for a term `h` for which this equality does not hold.