Concepts of Programming Language Design

Syntax Exercises

Liam O'Connor-Davis
Gabriele Keller

November 19, 2024

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:

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

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

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

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.

> **Solution:** This is one possible solution. It can be even simpler, as for the abstract syntax, it's not even necessary to distinguish between input and output. This then leads to expressions which are correct according to the abstract syntax, but don't correspond to correct expressions in the concrete syntax, but this is not a problem.
>
> $$\frac{x \in \{\texttt{a}, \texttt{b}\}}{x\ \textbf{Input}} \quad \frac{x \in \{\texttt{T}, \texttt{F}\}}{x\ \textbf{Output}} \quad \frac{c\ \textbf{Input} \quad t\ \textbf{A} \quad e\ \textbf{A}}{(\texttt{If}\ c\ t\ e)\ \textbf{A}} \quad \frac{x\ \textbf{Output}}{(\texttt{Out}\ x)\ \textbf{A}}$$

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

Solution:

$$\frac{}{\top \textbf{ Output}\leftrightarrow\texttt{(Out T)}}\textsc{Top} \qquad \frac{}{\bot \textbf{ Output}\leftrightarrow\texttt{(Out F)}}\textsc{Bot}$$

$$\frac{}{\alpha \textbf{ Input}\leftrightarrow\texttt{A}}\textsc{Input}_\alpha \quad \frac{}{\beta \textbf{ Input}\leftrightarrow\texttt{B}}\textsc{Input}_\beta$$

$$\frac{c \textbf{ Input} \quad \leftrightarrow c' \; t \textbf{ IExpr} \quad \leftrightarrow t' \; e \textbf{ Expr}\leftrightarrow e'}{\texttt{if } c \texttt{ then } t \texttt{ else } e \quad \textbf{Expr} \leftrightarrow \texttt{(If } c' \; t' \; e'\texttt{)}}\textsc{If}_1 \qquad \frac{c \textbf{ Input} \quad \leftrightarrow c' \; t \textbf{ IExpr} \quad \leftrightarrow t'}{\texttt{if } c \texttt{ then } t \textbf{ Expr}\leftrightarrow \texttt{(If } c' \; t'\texttt{) F}}\textsc{If}_2$$

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

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

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

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

Solution: (We drop the `Out` operator here to make the solution more compact)

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

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.

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

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

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

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

(b) Here is an incorrect substitution algorithm for this language:

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

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:

$$(\texttt{(Lambda}\ x\ e\texttt{)})[v := t] \quad \mapsto \quad \begin{cases} \texttt{(Lambda}\ x\ (e[v := t])\texttt{)} & \text{if } x \neq v \text{ and } x \notin FV(t) \\ \texttt{(Lambda}\ x\ e\texttt{)} & \text{if } x = v \\ \text{undefined} & \text{otherwise} \end{cases}$$

(c) Aside from the difficulties with substitution, using arbitrary strings for variable names in first-order abstract syntax means that 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.

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

   First order embedding:

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

   Higher order embedding:

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

   (a) Define a function of type

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

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

Solution:

```
data FOExpr
 = FONum  Int
 | FOVar  String
 | FOPlus FOExpr FOExpr
 | FOLet  String FOExpr FOExpr

data HOExpr
 = HONum  Int
 | HOPlus HOExpr HOExpr
 | HOLet  HOExpr (HOExpr -> HOExpr)
 | HODummy String -- we need this dummy variable during conversion


foToHO :: FOExpr -> HOExpr
foToHO foExpr = foToHO' foExpr []
  where
    foToHO' (FONum n) _ = HONum n
    foToHO' (FOVar str) env =
       case lookup str env of
         Just e  -> e
         Nothing -> error $ "expression not closed: var " ++
                       str ++ " occured freely"
    foToHO' (FOPlus e1 e2) env =
      HOPlus (foToHO' e1 env) (foToHO' e2 env)
    foToHO' (FOLet str e1 e2) env =
      HOLet (foToHO' e1 env) (\verb+\x+  -> foToHO' e2 ((str, x) : env))

hoToFO :: HOExpr -> FOExpr
hoToFO hoExpr = hoToFO' hoExpr 0
```

```
  where
    hoToFO' (HONum n) _            = FONum n
    hoToFO' (HODummy str) _        = FOVar str
    hoToFO' (HOPlus e1 e2) varCnt =
      FOPlus (hoToFO' e1 varCnt) (hoToFO' e2 varCnt)
    hoToFO' (HOLet e1 e2) varCnt  =
      FOLet varName (hoToFO' e1 varCnt)
                    (hoToFO' (e2 (HODummy varName)) (varCnt + 1))
        where
          varName = "x" ++ show varCnt


evalHO :: HOExpr -> Int
evalHO (HONum n)
  = n
evalHO (HOPlus e1 e2)
  = evalHO e1 + evalHO e2
evalHO (HOLet e1 e2)
  = evalHO (e2  e1)


evalFO :: FOExpr -> Int
evalFO foExpr = evalFO' foExpr []
  where
    evalFO' (FONum n) _ = n
    evalFO' (FOVar str) env =
      case lookup str env of
        Just v -> v
        Nothing -> error $ "expression not closed: var " ++ str ++
                     " occured freely"
    evalFO' (FOPlus e1 e2) env
      = evalFO' e1 env + evalFO' e2 env

    evalFO' (FOLet str e1 e2) env
      = evalFO' e2 ((str, evalFO' e1 env) : env)
```

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

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

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

```
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.

Solution: If the function in the higher-order let-binding expects the constructor, for example, and returns a differnt expression depending on the constructor:

```
HOLetBnd (HONum 3) f
  where
       f x = case x of
                    HONum n -> HONum (n+1)
                    _       -> x
```

Evaluating this expression results yields 4, but when converted to first-order and then back, it evaluates to 3.