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:

 $\overline{\top \text{ Output}} \quad \overline{\bot \text{ Output}} \quad \overline{\alpha \text{ Input}} \quad \overline{\beta \text{ Input}}$ $\frac{c \text{ Input } t \text{ IExpr } e \text{ Expr }}{\text{if } c \text{ then } t \text{ else } e \text{ Expr }} \quad \frac{c \text{ Input } t \text{ IExpr }}{\text{if } c \text{ then } t \text{ Expr }} \quad \frac{x \text{ Output }}{x \text{ IExpr }}$ $\frac{e \text{ Expr }}{(e) \text{ IExpr }} \quad \frac{e \text{ IExpr }}{e \text{ Expr }}$

If an else clause is omitted, the result of the expression if the condition is false is defaulted to \perp . For example, an AND or OR gate could be specified like so:

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

Or, a NAND gate:

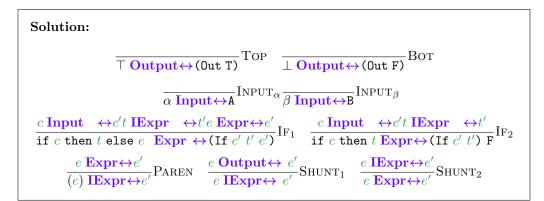
if α then (if β then \bot else $\top)$ else \top

(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.

 $\frac{x \in \{\mathbf{a}, \mathbf{b}\}}{x \text{ Input}} \quad \frac{x \in \{\mathbf{T}, \mathbf{F}\}}{x \text{ Output}} \quad \frac{c \text{ Input } t \mathbf{A} e \mathbf{A}}{(\text{If } c t e) \mathbf{A}} \quad \frac{x \text{ Output}}{(\text{Out } x) \mathbf{A}}$

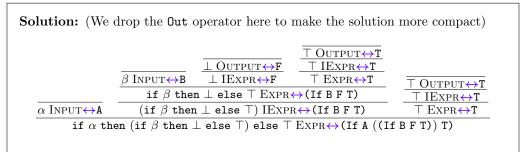
(b) Write rules for a parsing relation (\leftrightarrow) for this language.



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

	β Input \leftrightarrow		$ \begin{array}{c} \top \text{ Output} \leftrightarrow \\ \hline \top \text{ IExpr} \leftrightarrow \\ \hline \top \text{ Expr} \leftrightarrow \end{array} $	
	<u></u>		\uparrow $\Box APR \leftrightarrow$	\top Output \leftrightarrow
	$\texttt{if} \hspace{0.1cm} \beta \hspace{0.1cm} \texttt{then} \hspace{0.1cm} \bot \hspace{0.1cm} \texttt{else} \hspace{0.1cm} \top \hspace{0.1cm} \text{Expr} \longleftrightarrow$			\top IExpr \leftrightarrow
α Input \leftrightarrow	$(\texttt{if } \beta \texttt{ then } \bot \texttt{ else } \top) \text{ IExpr} \leftrightarrow$			$\top \text{Expr} \leftrightarrow$
if α then (if β then \perp else \top) else \top 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.

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

(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 ((Var x)))). Here, the name x is shadowed in the inner binding.

An $\alpha\text{-equivalent}$ 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:

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:

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

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

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 \mathbf{x} and \mathbf{y} for those function arguments. The only way for a Haskell function \mathbf{f} to be distinguished from a function \mathbf{g} is for $\mathbf{f} x$ to be different from $\mathbf{g} x$ 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+ \rightarrow 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)
                    (hoToF0' (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)
evalF0 :: F0Expr -> Int
evalF0 foExpr = evalF0' 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
      = evalF0' e1 env + evalF0' 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 ${\tt t}$ 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 \rightarrow HONum (n+1)
_ \rightarrow x
```

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