Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Algebraic structures; Logic programming Section and Practice Problems

Apr 7–8, 2016

1 Haskell

(a) Install the Haskell Platform, via `https://www.haskell.org/platform/`.

(b) Get familiar with Haskell. Take a look at `http://www.seas.harvard.edu/courses/cs152/2016sp/resources.html` for some links to tutorials.

In particular, get comfortable doing functional programming in Haskell. Write the factorial function. Write the append function for lists.

(c) Get comfortable using monads, and the bind syntax. Try doing the exercises at `https://wiki.haskell.org/All_About_Monads#Exercises` (which will require you to read the previous sections to understand `do` notation, and their previous examples).

(d) Also, look at the file `http://www.seas.harvard.edu/courses/cs152/2016sp/sections/haskell-examples.hs`, which includes some example Haskell code (that will likely be covered in Section).

2 Algebraic structures

(a) Show that the option type, with *map* defined as in the lecture notes (Lecture 18, Section 2.2) satisfy the functor laws.

Answer: *The functor laws are:*

$$\forall f \in A \to B, g \in B \to C.\ (map\ f) \,⨾\, (map\ g) = map\ (f \,⨾\, g) \qquad \textit{Distributivity}$$

$$map\ (\lambda a\!:\!A.\, a) = (\lambda a\!:\!T_A.\, a) \qquad \textit{Identity}$$

The definition of map for the option type is

$$map \equiv \lambda f\!:\!\tau_1 \to \tau_2.\, \lambda a\!:\!\tau_1\ \textbf{option}.\ \textsf{case}\ a\ \textsf{of}\ \lambda x\!:\!\textbf{unit}.\ \textsf{none} | \lambda v\!:\!\tau_1.\ \textsf{some}\ \ (f\ v)$$

To show distributivity, we need to show that for all functions $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_3$ we have that $\lambda x\!:\!\tau_1\ \textbf{option}.\ (map\ g)\ ((map\ f)\ x)$ is equivalent to $map\ (\lambda x\!:\!\tau_1.\, g\ (f\ x))$

Recall that ⨾ indicates function composition. So the function $f \,⨾\, g$ can can be expressed as $\lambda x\!:\!\tau_1.\, g\ (f\ x)$, and the function $(map\ f) \,⨾\, (map\ g)$ can be expressed as $\lambda x\!:\!\tau_1\ \textbf{option}.\ (map\ g)\ ((map\ f)\ x)$.

$\lambda x\!:\!\tau_1\ \textbf{option}.\ (map\ g)\ ((map\ f)\ x)$

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ (map\ g)\ ((\lambda a\!:\!\tau_1\ \textbf{option}.\ \textit{case}\ a\ \textit{of}\ \lambda y\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ (f\ v))\ x)$ *expand map f*

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ (map\ g)\ (\textit{case}\ x\ \textit{of}\ \lambda y\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ (f\ v))$ *by β-equivalence*

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ (\lambda b\!:\!\tau_2\ \textbf{option}.\ \textit{case}\ b\ \textit{of}\ \lambda z\!:\!\textbf{unit}.\ \textit{none}|\lambda w\!:\!\tau_2.\ \textit{some}\ \ (g\ w))$
$\quad (\textit{case}\ x\ \textit{of}\ \lambda y\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ (f\ v))$ *expand map g*

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ \textit{let}\ b = \textit{case}\ x\ \textit{of}\ \lambda y\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ (f\ v)\ \textit{in}$
$\quad \textit{case}\ b\ \textit{of}\ \lambda z\!:\!\textbf{unit}.\ \textit{none}|\lambda w\!:\!\tau_2.\ \textit{some}\ \ (g\ w)$ *rewrite as let expression*

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ \textit{case}\ x\ \textit{of}\ \lambda y\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ (g\ (f\ v))$ *simplifying nested cases*

$= \lambda x\!:\!\tau_1\ \textbf{option}.\ \textit{case}\ x\ \textit{of}\ \lambda t\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau_1.\ \textit{some}\ \ ((\lambda y\!:\!\tau_1.\ g\ (f\ y))\ v)$

$= map\ (\lambda y\!:\!\tau_1.\ g\ (f\ y))$ *un-expanding map $(\lambda y\!:\!\tau_1.\ g\ (f\ y))$*

To show identity, we need to show that map $\lambda a\!:\!\tau.\ a$ is equivalent to $\lambda a\!:\!\tau\ \textbf{option}.\ a$.

$map\ \lambda a\!:\!\tau.\ a$

$= \lambda b\!:\!\tau\ \textbf{option}.\ \textit{case}\ b\ \textit{of}\ \lambda x\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau.\ \textit{some}\ \ ((\lambda a\!:\!\tau.\ a)\ v)$ *expand map $(\lambda a\!:\!\tau.\ a)$*

$= \lambda b\!:\!\tau\ \textbf{option}.\ \textit{case}\ b\ \textit{of}\ \lambda x\!:\!\textbf{unit}.\ \textit{none}|\lambda v\!:\!\tau.\ \textit{some}\ \ v$ *by β-equivalence*

$= \lambda b\!:\!\tau\ \textbf{option}.\ b$

(b) Consider the list type, τ **list**. Define functions *return* and *bind* for the list monad that satisfy the monad laws. Check that they satisfy the monad laws.

Answer: *We define return and bind so that they represent a set of possible values that could be produced by a computation. That is, we use lists to represent the possible values of a nondeterministic computation. There are other ways to define return and bind on lists, for example, as a stream of results produced by a computation.*

Here return will take a value of type τ and return a list that contains the value as its only element. bind will take a list of τ, take a function f from τ to lists of τ', and apply f to every element of the list (using the function map, defined in class), to get a list of lists of τ'. We then use a utility function flatten to flatten the list of lists of τ' to a list of τ'. (In the definition of flatten, a acts as an accumulator.)

$return \triangleq \lambda x\!:\!\tau.\ x\ ::\ [\,]$

$bind \triangleq \lambda xs\!:\!\tau\ \textbf{list}.\ \lambda f\!:\!\tau \to \tau'\ \textbf{list}.\ \textit{flatten}\ (map\ f\ xs)$

$\textit{flatten} \triangleq \textit{let}\ fl = \mu f\!:\!\tau'\ \textbf{list} \to (\tau'\ \textbf{list})\ \textbf{list} \to \tau'\ \textbf{list}.$
$\quad \lambda a\!:\!\tau'\ \textbf{list}.\ \lambda x\!:\!(\tau'\ \textbf{list})\ \textbf{list}.\ \textit{if isempty?}\ x\ \textit{then}\ a\ \textit{else}\ f\ (append\ a\ (\textsf{head}\ x))\ (\textsf{tail}\ x)\ \textit{in}$
$\quad fl\ [\,]$

3 Logic Programming

(a) Consider the following Prolog program (where [] is a constant representing the empty list, $[t]$ is shorthand for $\mathsf{cons}(t, [])$ and $[t_1, t_2|t_3]$ is shorthand for $\mathsf{cons}(t_1, \mathsf{cons}(t_2, t_3))$.

$$\mathsf{foo}([], []).$$
$$\mathsf{foo}([X], [X]).$$
$$\mathsf{foo}([X, Y|S], [Y, X|T]) \text{ :– } \mathsf{foo}(S, T)$$

For each of the following queries, compute the substitutions that Prolog will generate, if any. (Note that there is a difference between an empty substitution, and no substitution.) If the query evaluation will not terminate, explain why.

- $\mathsf{foo}([\mathsf{a}, \mathsf{b}], X)$.
- $\mathsf{foo}([\mathsf{a}, \mathsf{b}, \mathsf{c}], X)$.
- $\mathsf{foo}([\mathsf{a}, \mathsf{b}], [\mathsf{a}, \mathsf{b}])$
- $\mathsf{foo}(X, [\mathsf{a}])$
- $\mathsf{foo}(X, Y)$.

Answer: *Intuitively,* $\mathsf{foo}(S, T)$ *holds for two lists S and T if they are the same length, and for all i, the $2i$th and $2i+1$th elements of S are equal, respectively, to the $2i+1$th and $2i$th elements of T.*

- $\mathsf{foo}([\mathsf{a}, \mathsf{b}], X)$.
  `X = [b,a]`
- $\mathsf{foo}([\mathsf{a}, \mathsf{b}, \mathsf{c}], X)$.
  `X = [b,a,c]`
- $\mathsf{foo}([\mathsf{a}, \mathsf{b}], [\mathsf{a}, \mathsf{b}])$
  No substitutions returned
- $\mathsf{foo}(X, [\mathsf{a}])$
  `X = [a]`
- $\mathsf{foo}(X, Y)$.
  ```
  X = [], Y = []
  X = [A, B], Y = [B, A]
  X = [A, B, C], Y = [B, A, C]
  X = [A, B, C, D], Y = [B, A, D, C]
  X = [A, B, C, D, E], Y = [B, A, D, C, E]
  X = [A, B, C, D, E, F], Y = [B, A, D, C, F, E]
  ```
  ...
  The evaluation of the query never terminates.

(b) Consider the following Datalog program.

$$\begin{aligned}
&\mathsf{bar(a,b,c)}.\\
&\mathsf{bar}(X,Y,Z) \coloneq \mathsf{bar}(Y,X,Z).\\
&\mathsf{bar}(X,Y,Z) \coloneq \mathsf{bar}(Z,Y,X), \mathsf{quux}(X,Z).\\
&\mathsf{quux(b,c)}.\\
&\mathsf{quux(c,d)}.\\
&\mathsf{quux}(X,Y) \coloneq \mathsf{quux}(Y,X).\\
&\mathsf{quux}(X,Z) \coloneq \mathsf{quux}(X,Y), \mathsf{quux}(Y,Z).
\end{aligned}$$

Find all solutions to the query $\mathsf{bar}(X,Y,Z)$.

Answer: *We start by the set of facts that are known, S_0, and then given S_i we produce S_{i+1} by unifying the horn clauses with the known facts to derive new facts, and repeat until we reach a fixed point.*

$$\begin{aligned}
S_0 = \{&\mathsf{bar(a,b,c)}.,\ \mathsf{quux(b,c)}.,\ \mathsf{quux(c,d)}.\}\\
S_1 = \{&\mathsf{bar(a,b,c)}.,\ \mathsf{quux(b,c)}.,\ \mathsf{quux(c,d)}., \mathsf{bar(b,a,c)}., \mathsf{quux(c,b)}., \mathsf{quux(d,c)}., \mathsf{quux(b,d)}.\}\\
S_2 = \{&\mathsf{bar(a,b,c)}.,\ \mathsf{quux(b,c)}.,\ \mathsf{quux(c,d)}., \mathsf{bar(b,a,c)}., \mathsf{quux(c,b)}., \mathsf{quux(d,c)}., \mathsf{quux(b,d)}.,\\
&\mathsf{bar(c,a,b)}., \mathsf{quux(d,b)}., \mathsf{quux(b,b)}.\mathsf{quux(c,c)}.\}\\
S_3 = \{&\mathsf{bar(a,b,c)}.,\ \mathsf{quux(b,c)}.,\ \mathsf{quux(c,d)}., \mathsf{bar(b,a,c)}., \mathsf{quux(c,b)}., \mathsf{quux(d,c)}., \mathsf{quux(b,d)}.,\\
&\mathsf{bar(c,a,b)}., \mathsf{quux(d,b)}., \mathsf{quux(b,b)}.\mathsf{quux(c,c)}., \mathsf{bar(a,c,b)}.\}\\
S_4 = \{&\mathsf{bar(a,b,c)}.,\ \mathsf{quux(b,c)}.,\ \mathsf{quux(c,d)}., \mathsf{bar(b,a,c)}., \mathsf{quux(c,b)}., \mathsf{quux(d,c)}., \mathsf{quux(b,d)}.,\\
&\mathsf{bar(c,a,b)}., \mathsf{quux(d,b)}., \mathsf{quux(b,b)}.\mathsf{quux(c,c)}., \mathsf{bar(a,c,b)}.\}
\end{aligned}$$

Since S_3 and S_4 are the same (i.e., applying the rules to S_3 doesn't derive any new facts) we have a fixed point. So all solutions to the query $\mathsf{bar}(X,Y,Z)$? *are:*

$$\begin{aligned}
&\mathsf{bar(a,b,c)}.\\
&\mathsf{bar(b,a,c)}.\\
&\mathsf{bar(c,a,b)}.\\
&\mathsf{bar(a,c,b)}.
\end{aligned}$$

4 Environment Semantics

For Homework 5, the monadic interpreter you will be using uses environment semantics, that is, the operational semantics of the language uses a map from variables to values instead of performing substitution. This is a quick primer on environment semantics.

An environment ρ maps variables to values. We define a large-step operational semantics for the lambda calculus using an environment semantics. A configuration is a pair $\langle e, \rho \rangle$ where expression e is the expression to compute and ρ is an environment. Intuitively, we will always ensure that any free variables in e are mapped to values by environment ρ.

The evaluation of functions deserves special mention. Configuration $\langle \lambda x.\, e, \rho \rangle$ is a function $\lambda x.\, e$, defined in environment ρ, and evaluates to the *closure* $(\lambda x.\, e, \rho)$. A closure consists of code along with values for all free variables that appear in the code.

The syntax for the language is given below. Note that closures are included as possible values and expressions, and that a function $\lambda x.\, e$ is *not* a value (since we use closures to represent the result of evaluating a function definition).

$$\begin{aligned} e &::= x \mid n \mid e_1 + e_2 \mid \lambda x.\, e \mid e_1\ e_2 \mid (\lambda x.\, e, \rho) \\ v &::= n \mid (\lambda x.\, e, \rho) \end{aligned}$$

Note than when we apply a function, we evaluate the function body using the environment from the closure (i.e., the lexical environment, ρ_{lex}), as opposed to the environment in use at the function application (the dynamic environment).

$$\frac{}{\langle x, \rho \rangle \Downarrow \rho(x)} \qquad \frac{}{\langle n, \rho \rangle \Downarrow n} \qquad \frac{\langle e_1, \rho \rangle \Downarrow n_1 \quad \langle e_2, \rho \rangle \Downarrow n_2}{\langle e_1 + e_2, \rho \rangle \Downarrow n}\ n = n_1 + n_2$$

$$\frac{}{\langle \lambda x.\, e, \rho \rangle \Downarrow (\lambda x.\, e, \rho)} \qquad \frac{\langle e_1, \rho \rangle \Downarrow (\lambda x.\, e, \rho_{lex}) \quad \langle e_2, \rho \rangle \Downarrow v_2 \quad \langle e, \rho_{lex}[x \mapsto v_2] \rangle \Downarrow v}{\langle e_1\ e_2, \rho \rangle \Downarrow v}$$

For convenience, we define a rule for let expressions.

$$\frac{\langle e_1, \rho \rangle \Downarrow (v_1, \rho) \quad \langle e_2, \rho[x \mapsto v_1] \rangle \Downarrow v_2}{\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \rho \rangle \Downarrow v_2}$$

(a) Evaluate the program $\mathsf{let}\ f = (\mathsf{let}\ a = 5\ \mathsf{in}\ \lambda x.\, a + x)\ \mathsf{in}\ f\ 6$. Note the closure that f is bound to.

(b) Suppose we replaced the rule for application with the following rule:

$$\frac{\langle e_1, \rho \rangle \Downarrow (\lambda x.\, e, \rho_{lex}) \quad \langle e_2, \rho \rangle \Downarrow v_2 \quad \langle e, \rho[x \mapsto v_2] \rangle \Downarrow v}{\langle e_1\ e_2, \rho \rangle \Downarrow v}$$

That is, we use the dynamic environment to evaluate the function body instead of the lexical environment.

What would happen if you evaluated the program $\mathsf{let}\ f = (\mathsf{let}\ a = 5\ \mathsf{in}\ \lambda x.\, a + x)\ \mathsf{in}\ f\ 6$ with this modified semantics?