CSCI 334: Principles of Programming Languages

Lecture 6: ML II

Instructor: Dan Barowy Williams

Announcements

HW3 is now out.
I will assume that you want to stay with your current partner. If this is not true, email me by tomorrow night and I will pair you with another student.

Announcements

midterm: before or after spring break?
“before” wins (by a lot)
Announcements

HW1 solutions handout
(fix: S2 is not worth 40 points!)

Announcements

Reminder: Thursday help (poorly attended)

Static vs. dynamic environments

fun add_one x = x + 1

What do we know about x?
What about 1?
What about add_one?

Static vs. dynamic environments

fun add_one x = x + 1

What do we know about x? int
What about 1? int; also 1
What about add_one? int -> int
Static vs. dynamic environments

fun add_one x = x + 1

Static environment:
Facts about a program that are always true.
E.g., data types.
Other static facts:
• “always halts”
• fn is named “add_one”

add_one 3

What do we know about x?
What about add_one 3?

x = 3
add_one 3 = 4

Static vs. dynamic environments

Dynamic environment:
Facts about a program that are true for a given invocation of the program.
E.g., values.
Other dynamic facts:
• “halts for given value”
Data type: a set of values and permissible operations on those values.

..., −1.1, −1.1, −1.0, 0, 1.0, 1.1, 1.11, ... ∈ real

+, −, <, >, <=, >=

Notice that = is not permitted

- 1.0 = 1.0;
stdIn:91.1-91.10 Error: operator and operand don't agree [equality type required]
operator domain: ''Z * ''Z
operand:         real * real
in expression:
    1.0 = 1.0

ML's uses a "structural type system"
Java uses a "nominal type system".
Structural Types

Types are equivalent if they have the same features. Base case in ML: same name; inductive case: same composition of names.

Matching names

val n = 3
val m = 4
n = m
false

Structural relationship

val a = (1,(2,"hi"))
val b = (1,(2,"hi"))
a = b
true

Matching names

val a = (1,(2,"hi"))
val b = (1,(2,"hi"))
a = b
true

map

This is essentially the same idea as in Lisp, but it is type-safe.

val xs = [1,2,3,4]
map (fn x => x + 1) xs
val xs = ["a","b","c"]
map (fn x => x + 1) xs
OK
Not OK

fold

Like map, in that it operates over lists, but only returns a single, “accumulated” object.

fun sum (l:int list):int = 
case l of
  [] => 0
  | x::xs => x + (sum xs)

fun concat (l:string list):string = 
case l of
  [] => ""
  | x::xs => x ^ (concat xs)

These look similar, no? Differences?

fold

Like map, in that it operates over lists, but only returns a single, “accumulated” object.

fun sum (l:int list):int = 
case l of
  [] => 0
  | x::xs => x + (sum xs)

fun concat (l:string list):string = 
case l of
  [] => ""
  | x::xs => x ^ (concat xs)

These look similar, no? Differences?
fun sum' (acc:int) (l:int list):int =  
  case l of  
    [] => acc  
    | x::xs => sum' (acc+x) xs

fun concat' (acc:string) (l:string list):string =  
  case l of  
    [] => acc  
    | x::xs => concat' (acc^x) xs

What is the function here?

fun f x y = x + y  
val f = fn : int -> int -> int
What is the type of the function that "abstracts over" the first and second \( e \)'s?

\[
\text{fun } f \, x \, y = x + y \\
\text{val } f = \text{fn : int } \to \text{ int } \to \text{ int}
\]

\[
\text{fun } f \, x \, y = x ^ \, y \\
\text{val } f = \text{fn : string } \to \text{ string } \to \text{ string}
\]

\[
\text{val } f = \text{fn : 'a } \to \text{ 'a } \to \text{ 'a}
\]

We now write a generic accumulation function.

\[
\text{fun } \text{foldl} \, (f: 'a*'b->'b) \, (acc: 'b) \, (l: 'a list): 'b = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => \text{foldl} \, f \, (f(x,acc)) \, xs
\]

\[
\text{fun sum'} \, (acc:int) \, (l:int list):int = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => \text{sum'} \, (acc+x) \, xs
\]

\[
\text{fun concat'} \, (acc:string) \, (l:string list):string = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => \text{concat'} \, (acc^x) \, xs
\]

\[
\text{fun foldl} \, (f: 'a*'b->'b) \, (acc: 'b) \, (l: 'a list): 'b = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => \text{foldl} \, f \, (f(x,acc)) \, xs
\]

\[
\text{fun foldr} \, (f:'a*'b->'b) \, (accum:'b) \, (l:'a list):'b = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => f(x, (foldr f acc xs))
\]

\[
\text{fun foldl} \, (f: 'a*'b->'b) \, (acc: 'b) \, (l: 'a list): 'b = \\
\text{case } l \text{ of} \\
\[ \] => acc \\
| x::xs => \text{foldl} \, f \, (f(x,acc)) \, xs
\]
Activities