CSCI 334: Principles of Programming Languages

Lecture 7: ML III

Instructor: Dan Barowy

Williams

Announcements

HW1 feedback and grade: look for pull request email

Official SML family reference: http://sml-family.org

Announcements

Homework help tonight 7-9pm (HERE)
**Intuition:**

Fold left

\[
\text{foldl (fn (x, acc) => acc+x) 0 [1,2,3,4]} \\
\text{acc = 0, [1,2,3,4]} \\
\text{acc = 0+1, [2,3,4]} \\
\text{acc = 1+2, [3,4]} \\
\text{acc = 3+3, [4]} \\
\text{acc 6+4, []} \\
\text{returns acc = 10}
\]

Fold right

\[
\text{foldr (fn (x, acc) => acc+x) 0 [1,2,3,4]} \\
\text{[1,2,3,4], acc = 0} \\
\text{[1,2,3], acc = 0+4} \\
\text{[1,2], acc = 4+3} \\
\text{[1], acc = 7+2} \\
\text{[], acc = 9+1} \\
\text{returns acc = 10}
\]

- Write a function `number_in_month` that takes a list of dates (where a date is `int*int*int`) and an `int` month and returns how many dates are in month
- Anyone try this at home?

fun `number_in_month` dates month = 
foldl (fn (_,m,_) => acc + (if m = month then 1 else 0)) 0 dates 
val `number_in_month` = 
fn : ('a * ''b * 'c) list -> ''b -> int
fold

structural recursion \rightarrow fold it!
(in a nutshell: any problem that recurses on a subset of input)

\[
\text{tree height} \quad \text{list length}
\]

evaluation

pattern matching

We’ve used this already. Did you notice?

\[
\text{fun number_in_month \ dates month : int = foldl (fn ((_,m,_),acc) => acc + (if m = month then 1 else 0)) 0 dates}
\]

pattern matching

A pattern is built from
• values,
• constructors,
• and variables

• Tests whether value(s) have shape defined by pattern
• If matches, binds variable(s) in pattern to value(s)

pattern matching

• Write a function \texttt{get_nth} that takes a list of strings and an int \texttt{n} and returns the \texttt{n}\textsuperscript{th} element of the list, where the head is \texttt{1}\textsuperscript{st}.
• Anyone try this at home?

\[
\text{fun get_nth \ nil \ n = NONE} \\
| \text{get_nth \ (x::xs) \ 1 = SOME \ x} \\
| \text{get_nth \ (x::xs) \ n = if n > 1 then get_nth \ xs \ (n-1) else NONE}
\]

val get_nth = fn : 'a list \rightarrow int \rightarrow 'a option
fun get_nth nil n = NONE
  | get_nth (x::xs) 1 = SOME x
  | get_nth (x::xs) n = if n > 1 then get_nth xs (n-1) else NONE

Another example: handling errors.

SML has exceptions (like Java)

But an alternative, easy way to handle many errors is to use the option type:

datatype 'a option = NONE | SOME of 'a

option type

Why option?

option is a data type;

not handling errors is a static type error!

Wait… isn’t this just the same thing as “checked exceptions” from Java?

They are similar but not the same. We’ll talk more about this in a coming lecture.
handling errors with patterns

- get_nth [1,2,3,4] 3
  val it = SOME 3 : int option
- get_nth [1,2,3,4] 0
  val it = NONE : int option
- get_nth [1,2,3,4] 5
  val it = NONE : int option

Let’s handle this error.
val i = ... whatever ...
val x = get_nth [1,2,3,4] i
case x of
  SOME n => print ((Int.toString n) ^ "\n")
| NONE   => print ((Int.toString i)
  ^ " is not a valid index!")

algebraic datatypes
(pattern matching’s best friend)

datatype 'a option =
  NONE
| SOME of 'a

datatype treat =
  SNICKERS
| TWIX
| TOOTSIE_ROLL
| DENTAL_FLOSS

fun trick_or_treat SNICKERS = “treat!”
| trick_or_treat TWIX = “treat!”
| trick_or_treat TOOTSIE_ROLL = “treat!”
| trick_or_treat DENTAL_FLOSS = “trick!”
algebraic datatypes  
(pattern matching’s best friend)

fun trick_or_treat SNICKERS = “treat!”
| trick_or_treat TWIX = “treat!”
| trick_or_treat TOOTSIE_ROLL = “treat!”
| trick_or_treat DENTAL_FLOSS = “trick!”

shorthand for case expression:

fun trick_or_treat t =
case t of
  SNICKERS => “treat!”
| TWIX => “treat!”
| TOOTSIE_ROLL => “treat!”
| DENTAL_FLOSS => “trick!”

Activity

Write a function is_older that takes two dates (where a date is int*int*int) and returns true or false. It evaluates to true if and only if the first argument is a date that comes before the second argument. If the two dates are the same, return false.

E.g.,
is_older (2018,2,21) (2018,2,22) returns true

Activity

Write a function num_before_sum that takes an int called sum (assume sum is positive) and an int list (assume all positive) and returns an int. The return value is an n such that the sum of the first n elements is < sum and the sum of the n + 1 elements is >= sum. Assume that the sum of the entire list > n. Summing goes from left to right.

E.g.,
num_before_sum 3 [0,1,2,3] returns 2