CSci 4223  
Principles of Programming Languages  
Lecture 3  
Prof. Clarkson  
Spring 2013

Leftover from last lecture
• Lists

Lists
• Despite nested tuples, the type of a variable still "commits" to a particular "amount" of data
• In contrast, a list can have any number of elements
• But unlike tuples, all elements have the same type
Need ways to build lists and access the pieces...

Building Lists
• The empty list is a value:
  
• In general, a list of values is a value; elements separated by commas:

• If e1 evaluates to v and e2 evaluates to a list [v1,...,vn], then e1::e2 evaluates to [v,...,vn]
  
• So [] can have type t list for any type t

Accessing Lists
Until we learn pattern-matching, we will use three standard-library functions
• null e evaluates to true if and only if e evaluates to []
• If e evaluates to [v1,v2,...,vn] then hd e evaluates to v1
  – raise exception if e evaluates to []
  – result is a list element
• If e evaluates to [v1,v2,...,vn] then tl e evaluates to [v2, ..., vn]
  – raise exception if e evaluates to []
  – result is a list

Type-checking list operations
Lots of new types: For any type t, the type t list describes lists where all elements have type t
• Examples int list bool list int list list (int * int) list (int list * int) list
• So [] can have type t list list for any type
• SML uses type 'a list to indicate this ("quote a" or "alpha")
• For el :: e2 to type check, we need a t such that el has type t and e2 has type t list. Then the result type is t list
• null :: 'a list -> bool
• hd :: 'a list -> 'a
• tl :: 'a list -> 'a list
Example list functions

```
fun sum_list (lst : int list) = 
  if null lst 
  then 0 
  else hd(lst) + sum_list(tl(lst))

fun countdown (x : int) = 
  if x=0 
  then [] 
  else x :: countdown (x-1)

fun append (lst1 : int list, lst2 : int list) = 
  if null lst1 
  then lst2 
  else hd (lst1) :: append (tl(lst1), lst2)
```

Recursion again

Functions over lists are usually recursive: only way to "get to" all the elements

- What should the answer be for the empty list?
- What should the answer be for a non-empty list?
  - Typically in terms of the answer for the tail of the list!

Similarly, functions that produce lists of potentially any size will be recursive
  - Create a list out of smaller lists

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
fun sum_pair_list (lst : (int*int) list) = 
  if null lst 
  then 0 
  else #1(hd lst) + #2(hd lst) + sum_pair_list(tl lst)

fun firsts (lst : (int*int) list) = 
  if null lst 
  then [] 
  else #1(hd lst) :: firsts(tl lst)

fun seconds (lst : (int*int) list) = 
  if null lst 
  then [] 
  else #2(hd lst) :: seconds(tl lst)

fun sum_pair_list2 (lst : (int*int) list) = 
  (sum_list (firsts lst)) + (sum_list (seconds lst))
```

Review

Huge progress on the core pieces of SML:
- Types: int bool unit t1*…*tn t list t1*…*tn->t
  - Types "nest" (each t above can be itself a compound type)
- Variables and environments
- Functions
  - Build: fun x0 (x1:t1, …, xn:tn) = e
  - Use: e0 (e1, …, en)
- Tuples
  - Build: (e1, …, en)
  - Use: #1 e, #2 e, …
- Lists
  - Build: [] e1::e2
  - Use: null e hd e tl e

Today

- A big thing we need: local bindings
  - For style and convenience
  - For efficiency (not "just a little faster")
  - A big but natural idea: nested function bindings
- A nice thing: options
  - ...finish all features necessary for homework 1
- Why not having mutation (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which Java programmers must obsess about

Let-expressions

The construct for introducing local bindings is just an expression, so we can use it anywhere we can use an expression

- Syntax: `let b1 b2 ... bn in e end`
  - Each bi is a binding and e is an expression
- Type-checking: Type-check each bi and e in a static environment that includes the previous bindings. Type of whole let-expression is the type of e.
- Evaluation: Evaluate each bi and e in a dynamic environment that includes the previous bindings. Result of whole let-expression is result of evaluating e.
Silly examples

fun silly1 (z : int) =
  let val x = if z > 0 then z else 3
  in
    if x > y then x*y else y*y
  end
fun silly2 () =
  let val x = 1
  in
    (let val x = 2 in x+1 end) +
    (let val y = x+a in y+1 end)
end

silly2 is poor style but shows let-expressions are expressions
– Could also use them in function-call arguments, parts of conditionals, etc.
– Also notice shadowing

What’s new

• What’s new is scope: where a binding is in the environment
  – In later bindings and body of the let-expression
    • (Unless a later or nested binding shadows it)
  – But not anywhere else

• Nothing else is new:
  – Can put any binding we want, even function bindings
  – Type-check and evaluate just like at “top-level”

Nested functions, part 1

• Good style to define helper functions inside the functions they help if they are
  – Unlikely to be useful elsewhere
  – Likely to be misused if available elsewhere
  – Likely to be changed or removed later

• A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later

(Inferior) Example

fun countup_from1 (x : int) =
  let fun count (from : int, to : int) =
    if from = to then to :: []
    else from :: count (from+1, to)
  in
count 1
end

Avoid repeated recursion

Consider this code and the recursive calls it makes
– Don’t worry about calls to null, hd, and tl because they do a small constant amount of work

fun bad_max (lst : int list) =
  if null lst
  then 0 (* horrible style; fix later *)
  else if null (tl lst)
    then hd lst
    else if hd lst > bad_max (tl lst)
      then hd lst
      else bad_max (tl lst)
  let x = bad_max [50,49,…,1]
  let y = bad_max [1,2,…,30]
end
Math never lies

Suppose one \texttt{bad\_max} calls if-then-else logic and calls to \texttt{hd} \texttt{null} \texttt{tl} take $10^{-7}$ seconds
- Then \texttt{bad\_max [50,49,...,1]} takes $50 	imes 10^{-7}$ seconds
- And \texttt{bad\_max [1,2,...,50]} takes $2.25 	imes 10^{8}$ seconds  
  (over 7 years)
- \texttt{bad\_max [55,54,...,1]} takes over 2 centuries

Buying a faster computer won't help much

The key is not to do repeated work that might do repeated work that might do...

- Saving recursive results in local bindings is essential...

---

Efficient max

\begin{verbatim}
fun good_max (list : int list) =
  if null list
     then 0 (* horrible style; fix later *)
     else if null (tl list)
            then hd list
            else let val tl_ans = good_max (tl list) in
                    if hd list > tl_ans
                    then hd list
                    else tl_ans
                    end
end
\end{verbatim}

---

Options

Having \texttt{good\_max} return 0 for the empty list is really awful
- Could raise an exception (see section this week)
- Could return a zero-element or one-element list
  - That works but is poor style because the built-in support for options expresses this situation directly

  - Type \texttt{t\_option} is a type for any type \texttt{t}
    - (much like \texttt{t\_list}, but a different type, not a list)

Building:
- \texttt{NONE} has type \texttt{\'a\_option} (much like \texttt{[]} has type \texttt{\'a\_list})
- \texttt{SOME e} has type \texttt{\'a\_option \# e has type \texttt{\'a}} (much like \texttt{e::[]})

Accessing:
- \texttt{isSome} has type \texttt{\'a\_option -> bool}
- \texttt{valOf} has type \texttt{\'a\_option -> \texttt{\'a}} (exception if given \texttt{NONE})

---

Example

\begin{verbatim}
fun better_max (list : int list) =
  if null list
     then NONE
     else let val tl_ans = better_max (tl list) in
            if isSome tl_ans
            and also valOf tl_ans > hd list
            then tl_ans
            else SOME (hd list)
            end
end
\end{verbatim}

---

Fast vs. fast

\begin{verbatim}
let val tl_ans = good_max (tl lst) in
  if hd lst > tl_ans
  then hd lst
  else tl_ans
end
\end{verbatim}

---

Fast vs. unusable

\begin{verbatim}
if hd lst > bad_max (tl lst)
  then hd lst
  else bad_max (tl lst)
\end{verbatim}
Example variation

```haskell
fun better_max2 (lst : int list) = 
    if null lst
    then NONE
    else let (* ok to assume lst nonempty b/c local *)
        fun max_nonempty (lst : int list) = 
            if null (tl lst)
            then hd lst
            else let val tl_ans = max_nonempty(tl lst)
                in 
                    if hd lst > tl_ans
                    then hd lst
                    else tl_ans
                end
            in 
                SOME (max_nonempty lst)
        end
end
```