Introduction to ML, III

Lars Birkedal

Programming, Logic and Semantics Group
IT University of Copenhagen

March, 2008
fun map f [] = []
  | map f (x::xr) = f x :: map f xr;

SML: anonymous functions

fun double x = 2 * x;

map double [4, 5, 89];

map (fn x => 2 * x) [4, 5, 89];
SML: the list iterator \( \text{foldr} \)

An application \( \text{foldr} \ f \ e \ xs \) replaces ‘::’ by \( f \) and \( [] \) by \( e \) in list \( xs \):

\[
\begin{align*}
\text{fun foldr} \ f \ e \ [] & = e \\
\text{ | foldr} \ f \ e \ (x::xr) & = f(x, \text{foldr} \ f \ e \ xr); \\
\end{align*}
\]

\[
\begin{align*}
\text{fun len} \ xs & = \text{foldr} \ (\text{fn} \ (_, \ res) \Rightarrow 1+\text{res}) \ 0 \ xs; \\
\text{fun sum} \ xs & = \text{foldr} \ (\text{fn} \ (x, \ res) \Rightarrow x+\text{res}) \ 0 \ xs; \\
\text{fun prod} \ xs & = \text{foldr} \ (\text{fn} \ (x, \ res) \Rightarrow x*\text{res}) \ 1 \ xs; \\
\text{fun map} \ g \ xs & = \text{foldr} \ (\text{fn} \ (x, \ res) \Rightarrow g \ x :: \text{res}) \ [] \ xs;
\end{align*}
\]

Using \( \text{foldr} \) and \( \text{map} \) to compute \( \text{freevars} \) for arbitrary primitives

\[
\begin{align*}
\text{fun freevars} \ e : \text{string list} = \\
\text{case} \ e \ \text{of} \\
\text{ | CstI} \ i & \Rightarrow [] \\
\text{ | Var} \ x & \Rightarrow [x] \\
\text{ | Let}(x, \ erhs, \ ebody) & \Rightarrow \\
\text{ | Prim}(\text{ope}, \ es) & \Rightarrow \text{foldr} \ \text{union} \ [] \ (\text{map} \ \text{freevars} \ es) \\
\end{align*}
\]
Inductive types

Lists are an example of an inductive type. The existence of `foldr` captures induction. More on the board.
fun tfold f e Lf = e
  | tfold f e (Br(v,t1,t2)) = f(v, tfold f e t1, tfold f e t2)

fun sumtree t = tfold (fn (v, r1, r2) => v + r1 + r2) 0 t
SML: the list iterator \texttt{foldl}

- fun reverse \(xs\) = \texttt{foldr (fn (x,ys) => ys @ [x]) []} \(xs\);
  > val 'b reverse = fn : 'b list -> 'b list
- reverse \([1,2,3]\);
  > val it = [3, 2, 1] : int list

- fun foldl (g: 'a -> 'b -> 'a) (acc: 'a) [] = acc
  | foldl g acc (x::xs) = foldl g (g acc x) xs;

- fun reverse' \(xs\) = foldl (fn ys => fn x => x::ys) [] \(xs\);
  > val 'b reverse' = fn : 'b list -> 'b list
- reverse' \([1,2,3]\);
  > val it = [3, 2, 1] : int list
Exercise

Use \texttt{foldr} to write function \texttt{filter}: \texttt{\textquotesingle}a\textquotesingle\ list \rightarrow (\texttt{\textquotesingle}a\textquotesingle\ \rightarrow \texttt{bool}) \rightarrow \texttt{\textquotesingle}a\textquotesingle\ list.
Homework presentation
Consider the following data type of terms:

```plaintext
datatype term =
  Var of int | Fun of string * term list;
```

exception Unify;

You should define a type

```plaintext
subst
```

for representing substitutions on terms and a function

```plaintext
apply_subst : subst -> term -> term
```

for applying a substitution to a term. Finally, you should write a function

```plaintext
unify : term * term -> subst
```

which, given two terms returns the most general unifier for the two terms; if no unifier exists you should raise the exception `Unify`.
Example, given the code below

```ml
val v0 = Var 0
val v1 = Var 1
val v2 = Var 2
val k = Fun("k", [])
val l = Fun("l", [])

val t1 = Fun("f", [v0,v1,k])
val t2 = Fun("f", [l,v2,k])

val S = unify(t1,t2)

val res = (apply_subst S t1) = (apply_subst S t2)
```

res should be bound to true.