7. Higher Order Operations on Lists

[Reade §3.4]

Examples.

(1) To ***double*** the values of a list of integers:

e.g. `double_list [3, 1, 4] = [6, 2, 8]`.

Def. by ***list recursion***:

```
fun double_list (a::x) = (2 * a)::(double_list x)
  | double_list [ ] = [ ];
```

Type?

- We want a ***more general*** (higher order) operation on lists, from which we can derive `double_list` as a special case!

(2) `map` $f\ [a_1, \ldots, a_n] \ = \ [f\,a_1, \ldots, f\,a_n]$ ("Map f over list")

```
fun map f [ ] = [ ]
  | map f (a::x) = (f a)::(map f x);
```

Type?

Then we can define

```
fun double x = 2 * x;

val double_list =
```

(3) To ***filter*** out of a list ℓ those items not satisfying a predicate p:

i.e., `filter` $p\ \ell$ $=$ sublist of ℓ containing those items which satisfy p.

```
fun filter p [ ] = [ ]
  | filter p (a::x) = if p a
                      then a::(filter p x)
                      else filter p x;
```

Definition by ***list recursion*** on ***2nd argument***.

Type?

(4) ***Quantifiers*** (*Transform predicates on T to predicates on T* `list`):

(a) $\quad \texttt{all}\ p\ \ell \ = \begin{cases} \texttt{true} & \text{if } \forall a \text{ in } \ell : p\,a \qquad (p \text{ is a predicate}) \\ \texttt{false} & \text{otherwise} \end{cases}$

```
fun all p (a::x) = (p a) andalso (all p x)
  | all p [ ]    =  ...
```

Def. is by . . . ?

Type?

(b) `exists` $p\ \ell$: $\quad$ **Ex**

(5) `addlists(`$[a_1$`, ... ,` $a_n]$`,` $[b_1$`, ... ,` $b_n]$`) =` $[a_1{+}b_1$`, ... ,` $a_n{+}b_n]$

```
fun addlists ([ ], [ ]) = [ ]
  | addlists (a::x, b::y) = (a + b)::(addlists (x, y))
```

! Warning: pattern matching is not exhaustive

Type?

Notes.

(1) Definition by ***list recursion*** on ***both arguments***.

(2) If arguments have different lengths: ***error***! (No pattern matching.)

- Again, we want a ***more general*** (higher order) operator:

(6) `zip` f $([a_1, \ldots, a_n],\ [b_1, \ldots, b_n]) \ = \ [\ f(a_1,\ b_1),\ \ldots,\ f(a_n,\ b_n)\ ]$

```
- fun zip f (a::x, b::y) = (f(a,b))::(zip f (x, y))
    | zip f ([], []) = [];
```

! Warning: pattern matching is not exhaustive

```
- zip plus ([2], [3,5]);
```

! Uncaught exception
! Match

Type?

Then we can define

```
val addlists = zip op+;
val multlists = zip op*;
```

etc.

Note: Again, get ***error*** with lists of different lengths.

Ex Define variants of `zip` which work with lists of different lengths:

(a) Append tail of longer list,

(b) Chop off tail of longer list.

(7) `quicksort : int list` $\rightarrow$ `int list` [See Reade p. 108]

Very nice! — it uses higher order operators (including `filter`).

(8) To ***sum*** a list of integers:

E.g. `sumlist [2, 6, 3] = 11`.

Two algorithms:

(i) ***Primitive*** (***list***) ***recursion***:

```
fun sumlist [ ] = 0
  | sumlist (b::x) = b + (sumlist x);
```

Type?

(ii) ***Tail (list) recursion***:

```
local fun sumlist_iter (a, [ ]) = a
        | sumlist_iter (a, b::x) = sumlist_iter (a + b, x)
 in fun   sumlist x = sumlist_iter (0, x)
 end;
```

Notes.

(1) Def. is by ***list recursion*** on ***2nd argt.***
Note the ***extra argument*** `a` in `sumlist_iter`.
(Think of `a` as the "partial sum", which *grows* from 0 to `sumlist`,
and `x` as the "partial list", which *shrinks* from the input list to [ ].)

(2) Iteration invariant $=$ `sumlist_iter (a, x)` $= \ldots$?
Bound value $= \ldots$? $\boxed{\textbf{Ex}}$

$\boxed{\textbf{Ex}}$ Define `prodlist` ℓ $=$ product of list ℓ of integers, analogous to (i) and (ii) above.

Again, we want a ***more general*** (***higher order***) operator.
We will generalise each of the above two definitions of `sumlist`.

(9)(i) Define `reduce` $f\ a\ \ell$ where

f is a binary function
a is ***starting value*** (for empty list)
ℓ is a list.

```
fun reduce f a [ ] = a
  | reduce f a (b::x) = f (b, (reduce f a x));
```

Note: Definition is by (***prim.***) ***list recursion*** on ***3rd argt***.

Type?

Hence:

(a) `val sumlist` $=$ `reduce` ...

(b) `val prodlist` $=$ `reduce` ...

(c) Let `flatten` be a function such that (e.g.)

```
flatten [[1, 2], [3], [4, 5, 6]] = [1, 2, 3, 4, 5, 6]
```

Ex `val flatten` $=$

Type?

Ex Define `minlist` ℓ $=$ minimum of a list ℓ of integers.
(If ℓ is empty, return an error message.)

(9)(ii) ***Accumulate***: `accum` $f\ a\ \ell$ is a ***tail recursive*** version of `reduce`.

(Again, think of a as a *partial result* which grows, and ℓ as a *partial list* which shrinks.)

```
fun accum f a [ ] = a
  | accum f a (b::x) = accum f (f (a, b)) x;
```

Type?

Note: We can define

`sumlist`,

`prodlist`,

`flatten`,

`minlist`, ...

from `accum` exactly as from `reduce` in (i), e.g.:

```
val sumlist = accum (op +) 0;
```

Informally ***compare*** these two functionals ('$\otimes$' is any binary operator in infix):

$$(i)\ \ \texttt{reduce}\ (\texttt{op}\ \otimes)\ a\ [\,b_1,\ b_2,\ b_3\,]\ =\ b_1 \otimes (b_2 \otimes (b_3 \otimes a))$$

$$(ii)\ \ \texttt{accum}\ (\texttt{op}\ \otimes)\ a\ [\,b_1,\ b_2,\ b_3\,]\ =\ ((a \otimes b_1) \otimes b_2) \otimes b_3$$

'`reduce`' is also called '`foldright`',

'`accum`' is also called '`foldleft`'.

Note. '`op` $\otimes$' is the ***prefix*** (uncurried) form of '$\otimes$'.

$\boxed{\textbf{Ex}}$ Give reasonable ***sufficient conditions*** on f and a such that for any list ℓ

$$\texttt{reduce}\ f\ a\ \ell\ =\ \texttt{accum}\ f\ a\ \ell.$$