A recursive function follows the structure of inductively-defined data.

With lists as our example, we shall study

1. inductive definitions (to specify data)
2. recursive functions (to process data)
3. frequent function templates
Specifying Types/Sets

Extensional

\{ n \mid n \text{ is a multiple of } 3 \}\n\{ p \mid p \text{ has red hair} \}\n
- defined by giving characteristics
- no info about how to generate elements

Intensional Let \( S \) be the smallest set of natural numbers satisfying

1. \( 0 \in S \),
2. \( x + 3 \in S \) whenever \( x \in S \).

- defined inductively
- describes how to generate elements
Why require \textit{smallest} solution?

Let \( S \) be a set of natural numbers satisfying

1. \( 0 \in S \),
2. \( x + 3 \in S \) whenever \( x \in S \).

Which sets \textit{satisfy} this specification?

- \( \{0, 3, 6, 9, \ldots\} \)
- \( \{0, 1, 3, 4, 6, 7, 9, 10, \ldots\} \)
- \( \ldots \)

By choosing the \textit{smallest} solution, we

- get \textit{exactly} those elements explicitly generated by the specification
- we can give a \textit{derivation} showing why each element belongs in the set.
Derivation of Set Elements

Let $S$ be the **smallest** set of natural numbers satisfying

1. $0 \in S$,
2. $x + 3 \in S$ whenever $x \in S$.

Example:

- $0 \in S$ (by rule 1)
- $3 \in S$ (by rule 2)
- $6 \in S$ (by rule 2)
- $9 \in S$ (by rule 2)

Non-example:

- $10$

Letting set be defined as the smallest gives us **constructive** information about the set.
BNF Inductive Specifications

Integer lists:

\(<\text{int-list}> \ ::= \ \text{nil} \mid \text{int} :: \text{int-list}\>

Example:

\[
1 :: 2 :: 3 :: \text{nil} \equiv [1, 2, 3]
\]

Derivation:

\[
\begin{align*}
\text{nil is an } \text{int-list} & \quad \text{(by rule 1)} \\
\Rightarrow \ 3 :: \text{nil is an } \text{int-list} & \quad \text{(by rule 2)} \\
\Rightarrow \ 2 :: 3 :: \text{nil is an } \text{int-list} & \quad \text{(by rule 2)} \\
\Rightarrow \ 1 :: 2 :: 3 :: \text{nil is an } \text{int-list} & \quad \text{(by rule 2)}
\end{align*}
\]

Note:

- recursion in grammar
- each use of :: increases list length by 1
Approximating Recursion

Grammar:

\[
\langle \text{int-list} \rangle ::= \text{nil} \mid \langle \text{int} \rangle :: \langle \text{int-list} \rangle
\]

We write a family of functions `list_sum_i`, with \( i \) the length of the argument:

```ocaml
fun list_sum_0(ls) = 0;

fun list_sum_1(ls) =
  hd(ls) + list_sum_0(tl(ls));

fun list_sum_2(ls) =
  hd(ls) + list_sum_1(tl(ls));

fun list_sum_3(ls) =
  hd(ls) + list_sum_2(tl(ls));

...  

- list_sum_3([1,2,3]);

val it = 6 : int
```
Putting It Together

We had

```plaintext
fun list_sum_0(ls) = 0;

fun list_sum_1(ls) = 
    hd(ls) + list_sum_0(tl(ls));

fun list_sum_2(ls) = 
    hd(ls) + list_sum_1(tl(ls));

fun list_sum_3(ls) = 
    hd(ls) + list_sum_2(tl(ls));

...  

Recursive function:

```plaintext
fun list_sum(ls) =
    if ls = nil 
    then 0 
    else hd(ls) + list_sum(tl(ls));
```
Using Patterns

For the grammar

\[
\textit<int\textendash list> ::= \textit{nil} \mid \textit{int} :: \textit<int\textendash list>
\]

we wrote

\[
\textbf{fun} \ \text{list\_sum}(\text{ls}) = \\
\quad \textbf{if} \ \text{ls} = \text{nil} \\
\quad \quad \textbf{then} \ 0 \\
\quad \quad \textbf{else} \ \text{hd}(\text{ls}) + \text{list\_sum}(\text{tl}(\text{ls}));
\]

but the correspondence is clearer by the ML patterns

\[
\textbf{fun} \ \text{list\_sum}(\text{ls}) = \\
\quad \textbf{case} \ \text{ls} \ \textbf{of} \\
\quad \quad \text{nil} \quad \Rightarrow \ 0 \\
\quad \quad \mid \ (n::\text{ns}) \Rightarrow n + \text{list\_sum}(\text{ns});
\]

or even better

\[
\textbf{fun} \ \text{list\_sum}(\text{nil}) = 0 \\
\quad \mid \ \text{list\_sum}(n::\text{ns}) = n + \text{list\_sum}(\text{ns});
\]
Recursion Template

Data Structure directly Function Structure

Grammar:

\[
\text{<int-list> ::= nil | <int> :: <int-list>}
\]

Template:

```
fun list_rec(nil) = ....
| list_rec(n::ns) = .... list_rec(ns)......;
```

Key points:

- for each case in BNF there is a case in function
- recursion occurs in function exactly where recursion occurs in BNF
- we may assume function “works” for sub-structures of the same type
More Examples

Add one to each element of list:

```
fun list_inc(nil) = nil
| list_inc(n::ns) = (n+1)::list_inc(ns);
```

Select those elements greater than five:

```
fun gt_five(nil) = nil
| gt_five(n::ns) =
  if n > 5
  then n::gt_five(ns)
  else gt_five(ns);
```

Append two lists:

```
fun append(nil, l2) = l2
| append(n::ns, l2) = n::append(ns, l2);
```
Map

Adding one to each element of list:

\[
\text{fun list\_inc (nil) = nil} \\
| \quad \text{list\_inc (n::ns) = (n+1)::list\_inc (ns);} \\
\]

Generalization: apply arbitrary function to each element

\[
\text{fun list\_map f nil = nil} \\
| \quad \text{list\_map f (n::ns) =} \\
\quad \quad \quad \quad \quad f(n)::list\_map f \ ns; \\
\]

Type of list\_map:

\[
fn : (\text{'}a \rightarrow \text{'}b) \rightarrow \text{'}a\ \text{list} \rightarrow \text{'}b\ \text{list} \\
\]

Instantiation: add one to each element

\[
\text{val my\_list\_inc = list\_map (fn} x \Rightarrow x + 1); \\
\]

Instantiation: square each element

\[
\text{val square\_list = list\_map (fn} x \Rightarrow x * x); \\
\]
Filter

Selecting only the elements greater than five:

```ocaml
fun gt_five (nil) = nil
| gt_five (n::ns) =
  if n > 5 then n::gt_five (ns)
  else gt_five (ns);
```

Generalization: select using arbitrary predicate

```ocaml
fun list_filter p nil = nil
| list_filter p (n::ns) =
  if p(n) then n::list_filter p ns
  else list_filter p ns;
```

Type of `list_filter`:

`('a -> bool) -> 'a list -> 'a list`

Instantiation: select those greater than five

```ocaml
val my_gt_five = list_filter (fn n => n > 5);
```

Instantiation: select the even elements

```ocaml
val evens = list_filter (fn n => n mod 2 = 0);
```
“Folding” all elements up by adding them

```ocaml
fun list_sum(nil) = 0
| list_sum(n::ns) = n + list_sum(ns);
```

**Generalization**: fold in arbitrary way

```ocaml
fun foldr f e nil = e
| foldr f e (x::xs) = f(x,(foldr f e xs))
```

Type of foldr:

```
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

**Instantiation**: my_addlist

```ocaml
fun my_addlist xs = foldr op+ 0 xs
```

**Instantiation**: my_identity

```ocaml
fun my_identity xs = foldr op:: nil xs
```

**Instantiation**: my_append

```ocaml
fun my_append xs ys = foldr op:: ys xs
```
Recall \texttt{foldr}, processing input from \textbf{right}:

\begin{verbatim}
fun foldr f e nil = e \\
| foldr f e (x::xs) = f(x, (foldr f e xs))
\end{verbatim}

Type of \texttt{foldr}:

\((\texttt{a} \times \texttt{b} \rightarrow \texttt{b}) \rightarrow \texttt{b} \rightarrow \texttt{a} \texttt{list} \rightarrow \texttt{b}\)

Example instantiation:

\texttt{foldl op:: nil xs}

which \textit{reverses} a list.
List Representation of Sets

Sets may be represented as lists

+ easy to code

? with or without duplicates

- not optimal for big sets

Testing membership:

```ml
- member [3,6,8] 4; val it = false : bool
- member [3,6,8] 6; val it = true : bool
```

Coding member:

```ml
fun member nil x = false
  | member (y::ys) x = if x = y then true else member ys x;
```
fun member nil x = false
| member (y::ys) x = if x = y then true else member ys x;

Type of member:

member = fn : 'a list -> 'a -> bool

Here double primes denotes an equality type.

- member [fn x => x+2, fn x => x+1] (fn x => x+1);

.. Error: operator and operand don’t agree
   [equality type required]
   operator domain: 'Z list
   operand: (int -> int) list

because functions cannot be tested for equality
Set Operations

Intersection:

fun intersect([], ys) = []
| intersect(x::xs, ys) =
  if member ys x
  then x :: intersect(xs, ys)
  else intersect(xs, ys);

Type of intersection:

''a list * ''a list -> ''a list

Union, with type

''a list * ''a list -> ''a list

fun union([], ys) = ys
| union(x::xs, ys) =
  if member ys x
  then union(xs, ys)
  else x :: union(xs, ys);
Removing Duplicates

```ocaml
fun remove_dups [] = [] | remove_dups (x::xs) =
    if member xs x
    then remove_dups xs
    else x :: remove_dups xs;

with type ''a list -> ''a list
```
We often want to associate keys with values. One way to do so is to maintain a list of pairs (key, value).

- easy to code
- not optimal for big sets

We want to write a lookup function

**Input** an association list, and a key

**Output** the value corresponding to the key

```haskell
fun lookup ((y, v)::ds) x =
  if x = y then v
  else lookup ds x

| lookup nil x = ???
```
Variants of Lookup

We may need to go for some rather arbitrary value that signals unsuccessful search:

```ml
fun lookup ((y, v)::ds) x = 
  if x = y then v
  else lookup ds x
| lookup nil x = ~1
```

Type of `lookup`:

```
('a * int) list -> 'a -> int
```

We thus lose some polymorphism. Instead, we may write

```ml
fun lookup nil x = NONE
| lookup ((y, v)::ds) x = 
  if x = y then SOME v
  else lookup ds x
```

Type of `lookup`:

```
('a * 'b) list -> 'a -> 'b option
```