Garbage collection

ML provides several ways to allocate & initialize new values:
(...), {...}, [...], ::
But it provides no way to deallocate/free values that are no longer being used

Instead, it provides automatic garbage collection:
when there are no more references to a value (either from variables or from other objects), it is deemed garbage, and the system will automatically deallocate the value

Evaluation of automatic garbage collection
+ dangling pointers impossible
  (could not guarantee safety without this!)
+ storage leaks impossible
+ simpler programming
+ can be more efficient!
  – less ability to carefully manage memory use & reuse

GCs exist even for C & C++, as free libraries

Garbage Collection: A Huge Idea

Pattern matching

Pattern matching: a convenient syntax for extracting components of compound values (tuple, record, or list)

A pattern looks like an expression to build a compound value, but with variable names to be bound in some places
• cannot use the same variable name more than once

Use pattern in place of variable on l.h.s. of val binding
• anywhere val can appear: either at top-level or in let

```plaintext
- val x = (false,17);
val a = false : bool
val b = 17 : int
val (root1, root2) = quad_roots(3.0,4.0,5.0);
val root1 = 0.786299647847 : real
val root2 = ~2.11963298118 : real
```

More patterns

```plaintext
- val [x,y] = 3::4::nil;
val x = 3 : int
val y = 4 : int

- val (x::y::zs) = [3,4,5,6,7];
val x = 3 : int
val y = 4 : int
val zs = [5,6,7] : int list

- val {name=n, age=x} =
  {age=18*3, name="joe " ^ "stevens");
val x = 54 : int
val n = "joe stevens" : string
```

Other kinds of patterns

Constants (ints, bools, strings, chars, nil) can be patterns:

```plaintext
- val ["hi", "there", name] =
  ["hi", "there", "bob");
val name = "bob" : string

- val (x,true,3, #"x",z) =
  (5.5, true, 3, #"x", [3,4]);
val x = 5.5 : real
val z = [3,4] : int list

- val (x::y::nil) = [3,4];
val x = 3 : int
val y = 4 : int
```

If don’t care about some component, can use a wildcard: _

```plaintext
- val (_::_::zs) = [3,4,5,6,7];
val zs = [5,6,7] : int list
```
Nested patterns

- `val {student={name=n, age=x}, grades=[g1, g2, g3], best_friends=[{name=f1, age=_}, {name=f2, age=_}]}::rest = {...};`

Arbitrary nesting of patterns is a consequence of orthogonality

Function argument patterns

Formal parameter of a fun declaration is a pattern

- `fun swap (a, b) = (b, a);`
  `val swap = fn : 'a*'b -> 'b*'a`
- `fun swap2 x = (#1 x, #2 x);`
  `val swap2 = fn : 'a*'b -> 'b*'a`
- `fun swap3 x = let val (a,b) = x in (b,a) end;`
  `val swap3 = fn : 'a*'b -> 'b*'a`

Arbitrary nesting of patterns is a consequence of orthogonality

Multiple cases

Often a function's implementation can be broken down into several different cases, based on the argument value

ML allows a single function to be declared via several cases
Each case identified using pattern-matching
- cases checked in order, until first matching case

- `fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2);`
  `val fib = fn : int -> int`

- `fun null nil = true
  | null (_::_) = false;`
  `val null = fn : 'a list -> bool`

- `fun append(nil, lst) = lst
  | append(x::xs, lst) = x :: append(xs, lst);`
  `val append = fn : 'a list * 'a list -> 'a list`

Defining Functions By Cases and Pattern-Matching: Big Ideas

Missing cases

What if we don’t provide enough cases?
- ML gives a warning message "match nonexhaustive" when function is declared (statically)
- ML raises an exception "nonexhaustive match failure" if invoked and no existing case applies (dynamically)

- `fun first Elem (x::xs) = x;
  Warning: match nonexhaustive
  x : x => ...
  val first Elem = fn : 'a list -> 'a`

- `fun first Elem [3, 4, 5];`
  `val it = 3 : int`

How would you provide an implementation of this missing case?
Exceptions

If get in a situation where you can’t produce a normal value of the right type, then can raise an exception
- aborts out of normal execution
- can be handled by some caller
- reported as a top-level “uncaught exception” if not handled

Step 1: declare an exception that can be raised
- exception EmptyList;

Step 2: use the raise expression where desired
- fun first_elem (x::xs) = x
  | first_elem nil = raise EmptyList;
val first_elem = fn : 'a list -> 'a
- first_elem [3,4,5];
val it = 3 : int
- first_elem [];
uncaught exception EmptyList

Handling exceptions

Add handler clause to expressions to handle (some) exceptions raised in that expression
- must return same type as handled expression

Syntax:
expr handle exn_name1 => expr;
  | exn_name2 => expr2
  ...
  | _ => expr

- fun second_elem l = first_elem (tl l);
val second_elem = fn : 'a list -> 'a
- (second_elem [3]
  handle EmptyList => ~1) + 5
val it = 4 : int

Exceptions with arguments

Can have exceptions with arguments
- exception IOError of int;

- (... raise IOError(-3) ...)
  handle IOError(code) => code ...