An extended example: binary trees

- Stores elements in sorted order
- Enables faster membership testing, printing out in sorted order

```plaintext
datatype 'a BTree =
    EmptyBTree
  | BTNode of 'a * 'a BTree * 'a BTree
```

First-class functions

- Can make code more reusable by parameterizing it by functions as well as values and types
- Simple technique: treat functions as first-class values
  - Function values can be created, used, passed around, bound to names, stored in other data structures, etc., just like all other ML values
  ```plaintext
  fun int_lt(x:int, y:int) = x < y;
  val int_lt = fn : int * int -> bool
  val int_lt(3,4); val it = true : bool
  val f = int_lt;
  val f = fn : int * int -> bool
  val f(3,4); val it = true : bool
  ```

Passing functions to functions

- A function can often be made more flexible if it takes another function as an argument
- Example:
  - Parameterize binary tree insert & member functions by the = and < comparisons to use
  - Parameterize the quicksort algorithm by the < comparison to use
  - Parameterize a list search function by the pattern being searched for

```plaintext
val find = fn : ('a -> bool) * 'a list -> 'a
fun is_good_grade(g) = g >= 90;
val is_good_grade = fn : int -> bool
val find(is_good_grade, [85,72,92,98,84]); val it = 92 : int
```

Binary tree functions, revisited

```plaintext
fun insert(x, EmptyBTree, eq, lt) =
    BTNode(x, EmptyBTree, EmptyBTree)
  | insert(x, n as BTNode(y,t1,t2), eq, lt) =
    if eq(x,y) then n
    else if lt(x,y) then
        BTNode(y, insert(x, t1, eq, lt), t2)
    else
        BTNode(y, t1, insert(x, t2, eq, lt))
val insert = fn : 'a * 'a BTree * ('a * 'a -> bool) * ('a * 'a -> bool) -> 'a BTree

fun member(x, EmptyBTree, eq, lt) = false
  | member(x, BTNode(y,t1,t2), eq, lt) =
    if eq(x,y) then true
    else if lt(x,y) then
        member(x, t1, eq, lt)
    else
        member(x, t2, eq, lt)
val member = fn : 'a * 'a BTree * ('a * 'a -> bool) * ('a * 'a -> bool) -> bool
```

Calling binary tree functions

```plaintext
val t = insert(5, EmptyBTree, op=, op<);
val t = BTNode (5,EmptyBTree,EmptyBTree)
val t = insert(2, t, op=, op<);
val t = insert(3, t, op=, op<);
val t = insert(7, t, op=, op<);
val it = true : bool
val it = false : bool
... definitions of person type, person_eq and person_lt functions, and p value
val pt = insert(p, EmptyBTree, person_eq, person_lt);
val pt = ... : person BTree
```
Storing functions in data structures

- It's a pain to keep passing around the `eq` and `lt` functions to all calls of `insert` and `member`!
- It's unreliable to depend on clients to pass in the right functions!
- Idea: store the functions in the tree itself

```ml
datatype 'a BT = EmptyBT | BTNode of 'a * 'a BT * 'a BT
fun ins(x, tree, eq, lt) = ...

fun mbr(x, tree, eq, lt) = ...
```

in

```ml
datatype 'a BTree = BTree of {tree:'a BT, eq:'a * 'a -> bool, lt:'a * 'a -> bool}
fun emptyBTree(eq,lt) = BTree{tree=EmptyBT, eq=eq, lt=lt}
fun insert(x, BTree{tree, eq, lt}) = BTree{tree=ins(x, tree, eq, lt), eq=eq, lt=lt}
fun member(x, BTree{tree, eq, lt}) = mbr(x, tree, eq, lt)
```

Records containing functions are ML's version of objects!

A common pattern: map

- Pattern: take a list and produce a new list, where each element of the output is calculated from the corresponding element of the input
- `map` captures this pattern
  ```ml
  map: ('a -> 'b) * 'a list -> 'b list
  ```
  [not quite the type of ML's predefined `map`; stay tuned]

- Example:
  - have a list of Fahrenheit temperatures for Seattle days
  - want to give a list of temps to friend in England
  ```ml
  fun f2c(f_temp) = (f_temp - 32.0) * 5.0/9.0;
  val f2c = fn : real -> real
  fun f_temps = [56.4, 72.2, 68.4, 78.4, 45.0];
  val f_temps = [56.4, 72.2, 68.4, 78.4, 45.0] : real list
  val c_temps = map(f2c, f_temps);
  ```

Another common pattern: filter

- Pattern: take a list and return the first element that passes some test (a predicate)
- `filter` captures this pattern
  ```ml
  filter: ('a -> bool) * 'a list -> 'a list
  ```
  [not quite the type of ML's predefined `filter`; stay tuned]

- Example: find first nice day
  ```ml
  fun is_nice_day(temp) = temp >= 70.0;
  val is_nice_day = fn : real -> bool
  val nice_days = filter(is_nice_day, f_temps);
  val nice_days = [72.2, 78.4] : real list
  ```

Anonymous functions

- Map functions and predicate functions often pretty simple, only used as argument to `map`, etc.;
- don't merit their own name
- Can directly write anonymous function expressions:
  ```ml
  fn patternformal => exprbody
  ```

Examples:

- `fn x => x + 1;`
- `val a_nice_day = find(is_nice_day, f_temps);` `a_nice_day = 72.2 : real`

Fun vs. fn

- `fn` expressions are a primitive notion
- `val` declarations are a primitive notion
- `fun` declarations are just a convenient syntax for `val + fn`
  ```ml
  fun f x => map
  ```
  a syntactic sugar for
  ```ml
  val f x = map
  ```
- `val` expressions are a primitive notion
  ```ml
  fun mortal x = x
  ```
  `fun` mortal x = x is syntactic sugar
  ```ml
  val mortal x = (fn x => x)
  ```

- Explains why the type of a `fun` declaration prints like a `val` declaration with a `fn` value:
  ```ml
  fun mortal x = x
  ```
  `val` mortal x = `fn x => x`

- Attributes of good design:
  - orthogonality of primitives
  - syntactic sugar for common combinations
Nested functions

A nested function can be written as a function returning another function. For example:

```ml
fun good_days(good_temp:real, temps:real list):real list =
  filter(fn temp => temp >= good_temp, temps);
val good_days = fn : real * real list -> real list
```

What’s interesting about the anonymous function expression `fn temp => temp >= good_temp`?

Nested functions and scoping

If functions can be written nested within other functions (whether named in a `let` expression, or anonymous) then can reference local variables in enclosing function scope.

Variables declared outside a scope are called free variables.

Makes nested functions a lot more useful in practice.

More than just hiding helper functions.

Beyond what can be done with function pointers in C/C++

C functions only have globals as free variables.

Akin to inner classes in Java.

Returning functions from functions

If functions are first-class, then should be able to create and return them.

Example: function composition

```ml
fun compose(f, g) = fn x => f(g(x));
val compose = fn : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Currying

A curried function takes some arguments and then computes & returns a function which takes additional arguments, without having to pass in the first arguments again.

Example: a curried version of `map`:

```ml
fun map f nil = nil
  | map f (x::xs) = f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list
```

Clean syntactic sugar for currying

- Allow multiple formal argument patterns — curried function
- Application ("function calling") written without parentheses
- Function type (->) associates right-to-left; higher precedence than infix operators
- Curried is the normal way to define ML functions
- ML’s predefined `map`, `filter`, and `find` are defined like this

First-class functions and scoping

- Lexical scoping is interesting if returning a function with free variables
- how to remember bindings of free variables?
- `let` binds local variables
- `val` binds function parameters

How are these two calls distinguished?

Where do bindings for `f` and `g` come from?

- All curried functions have free variables like this
- Many anonymous `fn` args (to `map` et al.) have free variables
Closures

- To support lexically nested procedures which can be returned out of their enclosing scope, must represent as a **closure**: a pair of code address and an **environment**
  - environment records bindings of free variables
  - closure no longer dependent on enclosing scope
  - pair and environment must be heap-allocated
  - e.g. ML, Scheme, Haskell, Smalltalk, Cecil

Restricted versions

- If only allow to pass nested procedures down, not return them, then can implement more cheaply
  - environment can be stack-allocated, not heap-allocated
    - e.g. Pascal, Modula-3
- If allow nested procedures but not first-class procedures, then cheaper still
  - do not need pair, just extra implicit environment argument
    - e.g. Ada
- If allow first-class procedures but no nesting, then can implement with just a code address
  - e.g. C, C++

A general pattern: fold

- The general pattern over lists simply abstracts the standard pattern of recursion
  - **Recursion pattern**:
    - fun f(…, nil, …)   = … (* base case *)
    - | f(…, x::xs, …) = … x … f(…, xs, …) … (* inductive case *)
  - Parameters of this pattern, for a list argument of type 'a list:
    - what to return as the base case result ('b)
    - how to compute the inductive result from the head and the recursive call ('a * 'b -> 'b)
  - **fold** captures this pattern
    - fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
      - iterate over elements left-to-right: **foldl**
      - iterate over elements right-to-left: **foldr**
      - for associative combining operators, order doesn’t matter
        - (which is the recursive pattern above?)

Examples using fold

- Summing all the elements of a list
  - val rainfall = [0.0, 1.2, 0.0, 0.4, 1.3, 1.1];
  - val total_rainfall = foldl (fn (rain, subtotal) => rain + subtotal) 0.0 rainfall;
    - val total_rainfall = 4.0 : real
  - Reusable **sum** function?

- What do these do?
  - foldl (fn(x,ls) => x::ls)nil [3,4,5];
  - foldr (fn(x,ls) => x::ls)nil [3,4,5];
  - foldr (fn(x,ls) => x::ls) [1,2,3] [4,5,6];

Polymorphic type inference

- ML infers types of expressions automatically, as follows:
  - design each declared variable & subexpression a fresh type variable
  - result of function is another type variable
  - where argument and return type variables across function cases
  - for each subexpression, generate constraints on types of its operands
    - constraints on type expressions must equal another
      - for subexpression, use polymorphic function, require specified type variables with fresh one for each subexpression
      - solve constraints by unifying type expressions
    - can partially fill in types, e.g.
      - (x+y) + (z+w) = (x+y+z+w)
        - fail for type constraints, e.g. 'a + 'b
  - If overloaded operator is unresolved after constraint solving, default to int version
  - Overconstrained (unsatisfiable constraints) ⇒ type error
  - Underconstrained (still some type variables) ⇒ a polymorphic result

Example #1

- fun sum lst =
  - if null lst then 0
  - else hd lst +
  - sum (tl lst)
### Example #2

```ml
fun map f nil     = nil
  | map f (x::xs) =
    f x ::
    map f xs
```

### Let-bound polymorphism

- ML type inference supports only **let-bound** polymorphism
  - only val/func-declared names can be polymorphic, not names of formals
  - implicit quantifiers of polymorphic variables are at outer level
    - `fun f x = x;`  
      ```ml
      val f = fn : 'a -> 'a
      ```
    - `fun f x = x;`  
      ```ml
      val f = fn : ('a->'a) -> 'a
      ```
    - `fun f x = x;`  
      ```ml
      val f = fn : ∀'a.'a->'a
      ```
  - What if ML allowed explicitly quantified polymorphic types for formals?
    ```ml
    fun g(f: ∀'a.'a->'a) = (f 3, f "hi")
    ```
  - Type inference precludes first-class polymorphic values

### Polymorphic vs. monomorphic recursion

- When analyzing the body of a polymorphic function, what do we do when we encounter a recursive call?
  - `fun f(x) = ...
    | f(hd(x)) ...
    | f(tl(x)) ...
  
- If allow **polymorphic recursion**, then `f` is considered polymorphic in body, and each recursive call uses a fresh instantiation (like any call to a polymorphic function)
  - Type inference under polymorphic recursion is **undecidable**
    - ML uses monomorphic recursion

### Nested polymorphic functions

- After doing type inference for a function, if any type variables remain in its type, then make the function polymorphic over them
  - But what about a nested function?
    - `fun g(u, v) = (
      [x,u],
      [v,v])
    `  
    ```ml
    val g = fn : ('a list * 'b list) -> 'a * 'b
    ```
    - `fun g(u, v) = (
      [x,u],
      [v,v])
    `  
    ```ml
    val g = fn : ('a list * 'b list) -> 'a * 'b
    ```
  - Type of `f`: `∀'a 'b . (null list -> int * string)

### Properties of ML type inference

- Hindley-Milner type inference
  - allows let-bound polymorphism only
  - universal parametric polymorphism
    - no constrained polymorphism (other than equality types)
  - Type inference yields **principal type** for expression
    - single most general type that can be inferred
  - Worst-case complexity of type inference: exponential time
  - Average case complexity: linear time

### References

- Support side-effects (mutation) through explicit reference values:
  ```ml
  val x = ref 0;
  ```
  ```ml
  val x = ref 0 : int ref
  ```
  - `ref` : `a -> 'a ref`
  - `ref` : `a ref -> 'a`
- Arrays: indexable mutable locations
  ```ml
  val x = ref 0 : int ref
  ```
- Must say which things are mutable
- Mutation is compartmentalized
References to polymorphic values?

- Try this:
  - fun id(x) = x;
  - val ID = fn : 'a -> 'a
  - val fp = ref id;
  (* error in real SML; pretend it's not *)
  - val fp = ref fn : ('a -> 'a) ref
  - (!fp true, !fp 5);
  (true, 5) : bool * int
  - fp := not;
  *hmmmm...
  - !fp 5
  CRASH!!!

The "value restriction"

- Cannot allow references to polymorphic values
- exception arguments similarly cannot be polymorphic
- In general, only polymorphic literals can be bound in val/func bindings, not polymorphic expressions
- get "non-generalizable type variable" error otherwise
- SML'90 had "weakly polymorphic types" instead

Functors

- Can parameterize structures by other structures
  - functor AListUser(AL:ASSOC_LIST) = struct
    - ... AL.store ... AL.fetch ...
  - end
  - only know aspects of AL that are defined by ASSOC_LIST
- Instantiate functors to build regular structures:
  - structure ALU1 = AListUser(Assoc_List);
  - structure ALU2 = AListUser(Hash_Assoc_List);

Functors for bounded quantification

- Define a signature representing the operations needed
  - signature ORDERED = sig
    - type T
    - val eq: T * T -> bool
    - val lt: T * T -> bool
  - end
- Define quantified algorithms as elements of functors parameterized by required signature
  - functor Sort(O:ORDERED) = struct
    - fun min(x,y) = if O.lt(x,y) then x else y
    - fun sort(lst) = ...
  - end

An instantiation of Sort

- Create specialized sorter by instantiating functor with appropriate operations
  - structure IntOrder:ORDERED = struct
    - type T = int
    - val eq = (op =)
    - val lt = (op <)
  - end
  - structure IntSort = Sort(IntOrder);
  - structure IntSort = ...
  - IntSort.sort([3,5,~2])
  - val it = [~2,3,5] : IntOrder.T list
- Use IntOrder:ORDERED, not IntOrder:>ORDERED
  - Using : instead of => allows type binding (T list) to bleed through to user of IntOrder
  - IntOrder is a view/extension of an existing type, int;
  - it isn't creating a new ADT x/only 2 operations

Another instantiation of Sort

- Can create nested, multiply parameterized functors:
  - functor PairOrder:
    - structure First:ORDERED;
    - structure Second:ORDERED;
  - STRUCTURE ORDERED = struct
    - type T = First.T * Second.T
    - val eq: T * T -> bool
    - val lt: T * T -> bool
  - end
  - (* to sort (int*string) lists: *)
  - structure IntStringSort = Sort(PreOrder(structure First = IntOrder;
    structure Second = StringOrder));
Signature “subtyping”

- Signature specifies a particular interface
- Any structure that satisfies that interface can be used where that interface is expected
  - e.g. in functor application
- Structure can have
  - more operations
  - more polymorphic operations
  - more details of implementation of types
  - than required by signature

Some limitations of ML modules

- Structures are not first-class values
  - must be named or be argument to functor application
- Structures must be declared at top level or nested inside another structure or signature
- Cannot instantiate functors at run-time to create “objects”
  ⇒ cannot simulate classes and object-oriented programming
- No type inference for functor arguments
- These constraints are to enable type inference of core and static typechecking (at all) of structures that contain types

Modules vs. classes

- Classes (abstract data types) implicitly define a single type, with associated constructors, observers, and mutators
- Modules can define 0, 1, or many types in same module, with associated operations over several types
  - no new types if adding operations to existing type(s)
    - e.g., a library of integer or array functions
  - hard to do in C++
- Modules can share private data & operations
- C++ requires names for all types (e.g. T)
  - class name is also type name in C++, conveniently
- Functors similar to parameterized classes
- C++’s public/private is simpler than ML’s separate signatures, but C++ doesn’t have a simple way of describing just an interface
- See Moby: modules + classes, cleanly