The problem is that we did a comparison in \texttt{merge} using the $\leq$ operator, and ML typed this as an integer comparison.

We can make our definition of sort more general by adding a comparison function, $\texttt{le}(a,b)$ as a parameter to \texttt{merge} and \texttt{sort}. If we curry this parameter we may be able to hide it from end users. Our updated definitions are:

\begin{verbatim}
fun merge(le, [], []) = []
| merge(le, [], hd::tl) = hd::tl
| merge(le, hd::tl, []) = hd::tl
| merge(le, hd::tl, h::t) = 
  if le(hd,h)
  then hd::merge(le, tl, h::t)
  else h::merge(le, hd::tl, t)
\end{verbatim}
fun sort le [] = []
  | sort le [a] = [a]
  | sort le (a::b::rest) = 
    let val (left,right) = 
      split(a::b::rest) in
      merge(le, sort le left, sort le right)
    end;

Now the types of `merge` and `sort` are:

val merge = fn :
  ('a * 'a -> bool) * 
   'a list * 'a list -> 'a list
val sort = fn : ('a * 'a -> bool) 
    -> 'a list -> 'a list

We can now “customize” `sort` by choosing a particular definition for the `le` parameter:

fun le(a,b) = a <= b;
val le = fn : int * int -> bool
fun intsort L = sort le L;
val intsort =
  fn : int list -> int list
intsort(
  [4,9,0,2,111,~22,8,~123]);
val it = [~123,~22,0,2,4,8,9,111] : int list

fun strle(a:string,b) =
  a <= b;
val strle =
  fn : string * string -> bool

fun strsort L = sort strle L;
val strsort =
  fn : string list -> string list
strsort(
  ["aac","aaa","ABC","123"]);
val it =
  ["123","ABC","aaa","aac"] : string list
Making the comparison relation an explicit parameter works, but it is a bit ugly and inefficient. Moreover, if we have several functions that depend on the comparison relation, we need to ensure that they all use the same relation. Thus if we wish to define a predicate \texttt{inOrder} that tests if a list is already sorted, we can use:

\begin{verbatim}
fun inOrder le [] = true
  | inOrder le [a] = true
  | inOrder le (a::b::rest) = le(a,b) andalso inOrder le (b::rest);
\end{verbatim}

Now \texttt{sort} and \texttt{inOrder} need to use the same definition of \texttt{le}. But how can we enforce this?
The structure mechanism we studied earlier can help. We can put a single definition of `le` in the structure, and share it:

```
structure Sorting =
struct
    fun le(a,b) = a <= b;

    fun split [] = ([],[])
    | split [a] = ([a],[])
    | split (a::b::rest) =
        let val (left,right) =
            split rest in
            (a::left,b::right)
        end;

    fun merge([],[]) = []
    | merge([],hd::tl) = hd::tl
    | merge(hd::tl,[]) = hd::tl
    | merge(hd::tl,h::t) =
        if le(hd,h)
        then hd::merge(tl,h::t)
        else h::merge(hd::tl,t)
```
fun sort [] = []
  | sort([a]) = [a]
  | sort(a::b::rest) = 
    let val (left,right) = 
      split(a::b::rest) in 
      merge(sort(left), 
        sort(right))
    end;

fun inOrder [] = true
  | inOrder [a] = true
  | inOrder (a::b::rest) = 
    le(a,b) andalso 
    inOrder (b::rest);
end;

structure Sorting : 
sig 
  val inOrder : int list -> bool 
  val le : int * int -> bool 
  val merge : int list * 
    int list -> int list 
  val sort : 
    int list -> int list 
end
val split : 'a list -> 'a list * 'a list
end

To sort a type other than integers, we replace the definition of \texttt{le} in the structure.

But rather than actually edit that definition, ML gives us a powerful mechanism to parameterize a structure. This is the \texttt{functor}, which allows us to use one or more structures as parameters in the definition of a structure.
Functors

The general form of a functor is

functor name
    (structName:signature) =
    structure definition;

This functor will create a specific version of the structure definition using the structure parameter passed to it.

For our purposes this is ideal—we pass in a structure defining an ordering relation (the \( \leq \) function). This then creates a custom version of all the functions defined in the structure body, using the specific \( \leq \) definition provided.
We first define
signature Order =
sig
  type elem
  val le : elem*elem -> bool
end;
This defines the type of a structure that defines a le predicate defined on a pair of types called elem.
An example of such a structure is
structure IntOrder:Order =
struct
  type elem = int;
  fun le(a,b) = a <= b;
end;
Now we just define a functor that creates a Sorting structure based on an Order structure:
functor MakeSorting(O:Order) =
struct
  open O; (* makes le available*)
  fun split [] = ([],[])
    | split [a] = ([a],[]) 
    | split (a::b::rest) =
      let val (left,right) = split rest in
        (a::left,b::right)
      end;

  fun merge([],[]) = []
    | merge([],hd::tl) = hd::tl
    | merge(hd::tl,[]) = hd::tl
    | merge(hd::tl,h::t) =
      if le(hd,h)
        then hd::merge(tl,h::t)
        else h::merge(hd::tl,t)
end;
fun sort [] = []
  | sort([a]) = [a]
  | sort(a::b::rest) = 
    let val (left,right) = 
      split(a::b::rest) in 
      merge(sort(left),
           sort(right))
    end;

fun inOrder [] = true
  | inOrder [a] = true
  | inOrder (a::b::rest) = 
    le(a,b) andalso
    inOrder (b::rest);
  end;
Now

structure IntSorting = 
    MakeSorting(IntOrder);

creates a custom structure for sorting integers:

    IntSorting.sort [3,0,~22,8];
    val it = [~22,0,3,8] : elem list

To sort strings, we just define a structure containing an \texttt{le} defined for strings with \texttt{Order} as its signature (i.e., type) and pass it to \texttt{MakeSorting}:

structure StrOrder:Order = 
    struct
        type elem = string
        fun le(a:string,b) = a <= b;
    end;
structure StrSorting = 
   MakeSorting(StrOrder);

StrSorting.sort(  
   ["cc","abc","xyz"]);

val it = ["abc","cc","xyz"] : StrOrder.elem list

StrSorting.inOrder(  
   ["cc","abc","xyz"]);

val it = false : bool

StrSorting.inOrder(  
   [3,0,~22,8]);

stdIn:593.1-593.32 Error:  
operator and operand don’t agree  
[literal]

   operator domain: strOrder.elem list

   operand:       int list

   in expression:  
   StrSorting.inOrder (3 :: 0 ::  
   ~22 :: <exp> :: <exp>)}
The SML Basis Library

SML provides a wide variety of useful types and functions, grouped into structures, that are included in the Basis Library.

A web page fully documenting the Basis Library is linked from the ML page that is part of the Programming Languages Links page on the CS 538 home page.

Many useful types, operators and functions are “preloaded” when you start the SML compiler. These are listed in the “Top-level Environment” section of the Basis Library documentation.

Many other useful definitions must be explicitly fetched from the structures they are defined in.
For example, the Math structure contains a number of useful mathematical values and operations. You may simply enter

open Math;
while will load all the definitions in Math. Doing this may load more definitions than you want. What’s worse, a definition loaded may redefine a definition you currently want to stay active. (Recall that ML has virtually no overloading, so functions with the same name in different structures are common.)

A more selective way to access a definition is to qualify it with the structure’s name. Hence

Math.pi;
val it = 3.14159265359 : real
gets the value of \texttt{pi} defined in \texttt{Math}. Should you tire of repeatedly qualifying a name, you can (of course) define a local value to hold its value. Thus

\begin{verbatim}
val pi = Math.pi;
val pi = 3.14159265359 : real
\end{verbatim}
works fine.
An Overview of Structures in the Basis Library

The Basis Library contains a wide variety of useful structures. Here is an overview of some of the most important ones.

- Option
  Operations for the `option` type.
- Bool
  Operations for the `bool` type.
- Char
  Operations for the `char` type.
- String
  Operations for the `string` type.
- Byte
  Operations for the `byte` type.
- Int
  Operations for the `int` type.
• **IntInf**
  Operations for an unbounded precision integer type.

• **Real**
  Operations for the real type.

• **Math**
  Various mathematical values and operations.

• **List**
  Operations for the list type.

• **ListPair**
  Operations on pairs of lists.

• **Vector**
  A polymorphic type for immutable (unchangeable) sequences.

• **IntVector, RealVector, BoolVector, CharVector**
  Monomorphic types for immutable sequences.
• Array
  A polymorphic type for mutable (changeable) sequences.

• IntArray, RealArray, BoolArray, CharArray
  Monomorphic types for mutable sequences.

• Array2
  A polymorphic 2 dimensional mutable type.

• IntArray2, RealArray2, BoolArray2, CharArray2
  Monomorphic 2 dimensional mutable types.

• TextIO
  Character-oriented text IO.

• BinIO
  Binary IO operations.

• OS, Unix, Date, Time, Timer
  Operating systems types and operations.
ML Type Inference

One of the most novel aspects of ML is the fact that it infers types for all user declarations.

How does this type inference mechanism work?

Essentially, the ML compiler creates an unknown type for each declaration the user makes. It then solves for these unknowns using known types and a set of type inference rules. That is, for a user-defined identifier $i$, ML wants to determine $T(i)$, the type of $i$. 
The type inference rules are:

1. The types of all predefined literals, constants and functions are known in advance. They may be looked-up and used. For example,

   \[
   2 : \text{int} \\
   \text{true} : \text{bool} \\
   [] : 'a \text{list} \\
   :: : 'a * 'a \text{list} \rightarrow 'a \text{list}
   \]

2. All occurrences of the same symbol (using scoping rules) have the same type.

3. In the expression

   \[ I = J \]

   we know \( T(I) = T(J) \).
4. In a conditional
   \((\text{if } E_1 \text{ then } E_2 \text{ else } E_3)\)
   we know that
   \[ T(E_1) = \text{bool}, \]
   \[ T(E_2) = T(E_3) = T(\text{conditional}) \]

5. In a function call
   \((f \ x)\)
   we know that if \(T(f) = 'a \rightarrow 'b\)
   then \(T(x) = 'a\) and \(T(f \ x) = 'b\)

6. In a function definition
   \[
   \text{fun } f \ x = \text{expr};
   \]
   if \(t(x) = 'a\) and \(T(\text{expr}) = 'b\)
   then \(T(f) = 'a \rightarrow 'b\)

7. In a tuple \((e_1, e_2, \ldots, e_n)\)
   if we know that \(T(e_i) = 'a_i \ 1 \leq i \leq n\)
   then \(T(e_1, e_2, \ldots, e_n) = 'a_1*'a_2*\ldots*'a_n\)
8. In a record

\{ a=e_1, b=e_2, \ldots \}

if \( T(e_i) = 'a_i \ 1 \leq i \leq n \) then

the type of the record =

\{a:'a_1, b:'a_2, \ldots\}

9. In a list \([v_1, v_2, \ldots v_n]\)

if we know that \( T(v_i) = 'a_i \ 1 \leq i \leq n \)

then we know that

'\(a_1 = 'a_2 = \ldots = 'a_n\) and

\(T([v_1, v_2, \ldots v_n]) = 'a_1\) list
To Solve for Types:

1. Assign each untyped symbol its own distinct type variable.
2. Use rules (1) to (9) to solve for and simplify unknown types.
3. Verify that each solution “works” (causes no type errors) throughout the program.

Examples

Consider

\[
\text{fun fact}(n)=
\begin{align*}
\text{if } n=1 \text{ then } 1 & \text{ else } n*\text{fact}(n-1); \\
T(\text{fact}) &= \text{ 'a} \rightarrow \text{ 'b} \\
T(n) &= \text{ 'c}
\end{align*}
\]

To begin, we’ll assign type variables:
Now we begin to solve for the types 'a, 'b and 'c must represent.

We know (rule 5) that 'c = 'a since n is the argument of fact.

We know (rule 3) that 'c = T(1) = int since n=1 is part of the definition.

We know (rule 4) that T(1) = T(if expression) = 'b since the if expression is the body of fact.

Thus, we have

'a = 'b = 'c = int, so

T(fact) = int -> int

T(n) = int

These types are correct for all occurrences of fact and n in the definition.
A Polymorphic Function:

fun leng(L) = 
  if L = []
  then 0
  else 1+len(tl L);
To begin, we know that
T([]) = 'a list and
T(tl) = 'b list -> 'b list
We assign types to leng and L:
T(leng) = 'c -> 'd
T(L) = 'e
Since L is the argument of leng,
'e = 'c
From the expression L=[] we know
'e = 'a list
From the fact that 0 is the result of the then, we know the if returns an int, so 'd = int.

Thus \( T(\text{leng}) = \text{'a list -> int} \) and \( T(L) = \text{'a list} \)

These solutions are type correct throughout the definition.