Lists

Lars-Henrik Eriksson

Functional Programming 1

Based on a presentation by Tjark Weber and notes by

Pierre Flener, Jean-Noël Monette, Sven-Olof Nyström

Today

- Polymorphism
- Lists: Motivation, Definition, Examples
- Pattern Matching and Recursion for Lists
- Example: Merge Sort
- List Operations in the SML Basis Library

Polymorphism

Motivation: Code Re-Use Across Types

Suppose we want to swap the components in a pair of integers.

```
fun swap (x, y) = (y, x)

> swap (1, 2);
val it = (2, 1): int * int
```

Motivation: Code Re-Use Across Types

Suppose we want to swap the components in a pair of integers.

```
fun swap (x, y) = (y, x)

> swap (1, 2);
val it = (2, 1): int * int
```

Nothing in `swap` is specific to integers. How about swapping the components in a pair of strings?

```
> swap ("a", "b");
val it = ("b", "a"): string * string
```

Or in a pair whose components are of different types?

```
> swap (1, "b");
val it = ("b", 1): string * int
```

`swap` Could Have Many Types ...

It is great that we only need to write `swap` once, and can then use it for arguments of different types.

But what is the type of `swap`?

It is not `int*int -> int*int`, because then we couldn't apply `swap` to pairs of strings. Likewise, it is not `string*string -> string*string`, because then we couldn't apply `swap` to pairs of integers.

Types may contain **type variables**: e.g., α, β, ...

Polymorphic Types

Types that contain type variables are known as **polymorphic** types.

Expressions whose type is polymorphic are also called polymorphic: e.g., swap is a polymorphic function.

In SML (where we don't have Greek characters), type variables are written with a ' as their first symbol: e.g., 'a, 'b, 'foo, .

```
> fun swap (x, y) = (y, x);
val swap = fn: 'a * 'b -> 'b * 'a
```

Examples of Polymorphic Expressions

There are many examples of polymorphic functions. Later, we will also see polymorphic values that are not functions.

```
> fun id x = x;
```

Examples of Polymorphic Expressions

There are many examples of polymorphic functions. Later, we will also see polymorphic values that are not functions.

```
> fun id x = x;
val id = fn: 'a -> 'a

> fun fst (x, _) = x;
```

Examples of Polymorphic Expressions

There are many examples of polymorphic functions. Later, we will also see polymorphic values that are not functions.

```
> fun id x = x;
val id = fn: 'a -> 'a

> fun fst (x, _) = x;
val fst = fn: 'a * 'b -> 'a

> fun snd (_, y) = y;
```

Examples of Polymorphic Expressions

There are many examples of polymorphic functions. Later, we will also see polymorphic values that are not functions.

```
> fun id x = x;
val id = fn: 'a -> 'a

> fun fst (x, _) = x;
val fst = fn: 'a * 'b -> 'a

> fun snd (_, y) = y;
val snd = fn: 'a * 'b -> 'b
```

Instantiation of Type Variables

When polymorphic expressions are used, their type variables are **instantiated** (i.e., replaced with types) if necessary to obtain a type-correct expression.

```
> id;
val it = fn: 'a -> 'a
> id 1;
val it = 1: int
> id "foo";
val it = "foo": string
> (id 1, id "foo");
val it = (1, "foo"): int * string
```

SML's Value Restriction

For technical reasons, a declaration **val** x = e can give `x` a polymorphic type only if `e` is a value (e.g., a constant, identifier, function expression `fn x' => e'`, tuple of values, .). This is known as the **value restriction**. Notably, function calls are not values.

```
> val id = fn x => x;
val id = fn: 'a -> 'a

> val mono1 = let in fn x => x end;
Warning [...]
val mono1 = fn: _a -> _a

> val mono2 = id swap;
Warning [...]
val mono2 = fn: _a * _b -> _b * _a
```

Polymorphism vs. Overloading

Overloading (sometimes also called *ad-hoc polymorphism*) allows to provide different functions of the same name for a limited number of individually specified types.

For instance, SML provides different implementations of `<` for types `int`, `real` and `string`.

Polymorphism (sometimes also called *parametric polymorphism*) allows code that does not mention any specific type to be used transparently with *any* type.

For instance, `(fn x => x)` can be applied to arguments of any type.

Equality and Polymorphism

What is the type of equality (=) in SML?

Equality and Polymorphism

What is the type of equality (=) in SML?

```
> op=;
val it = fn: ''a * ''a -> bool
```

Note the *two* quotation marks! These denote **equality types**.

Why is the type of `=` not simply `'a * 'a -> bool`?

Equality and Polymorphism

What is the type of equality (=) in SML?

```
> op=;
val it = fn: ''a * ''a -> bool
```

Note the *two* quotation marks! These denote **equality types**.

Why is the type of `=` not simply `'a * 'a -> bool`?

Because then `=` could be applied to arguments of *any* type. But certain types (e.g., `real`, function types) do not provide an equality test.

Equality Types

Equality types include

- `bool`, `char`, `int`, `string`, `unit`,
- `T1 * ... * Tn`, provided all of `T1`, ., `Tn` are equality types,
- data types (e.g., lists), under certain conditions.

Notably, `real` and `X -> Y` (for any types `X`, `Y`) are *not* equality types.

```
> (op =): int * int -> bool;
val it = fn: int * int -> bool
> (op =): real * real -> bool;
Error-Type mismatch in type constraint.
   Value: (=) : ''a * ''a -> bool
   Constraint: real * real -> bool
   Reason: Can't unify ''a to real (Requires equality type)
Found near (=) : real * real -> bool
```

Equality Types and Type Inference

If the equality (or inequality) operator is applied to polymorphic expressions, SML will correctly infer that these expressions have to be of equality type.

```
> fun silly (a, b, c, d, e) = a=b orelse d<>e;
```

Equality Types and Type Inference

If the equality (or inequality) operator is applied to polymorphic expressions, SML will correctly infer that these expressions have to be of equality type.

```
> fun silly (a, b, c, d, e) = a=b orelse d<>e;
val silly = fn: ''a * ''a * 'b * ''c * ''c -> bool
```

Lists:

Motivation, Definition, Examples

Tuples Have a Fixed Number of Components

When you use tuples in your program, you have to choose the number of components *at the time you write your code*.

For instance, you can write a function that computes the maximum of two integers

fun max2 (a, b) = **if** a > b **then** a **else** b

and another function that computes the maximum of three integers

fun max3 (a, b, c) = ...

and so on, but with tuples you cannot write *one* function that computes the maximum of an arbitrary number of integers.

Lists in SML: Examples

```
[18, 12, ~5, 12, 10] : int list

[2.0, 5.3 - 1.2, 3.7, ~1E5] : real list

["Bread", "Butter", "Milk"] : string list

[(1,"A"), (2,"B")] : (int*string) list

[Math.sin, fn x=>x+1.0] : (real->real) list

[[1], [2, 3]] : int list list

[] : 'a list
```

SML's `list` Type

Lists in SML must be **homogeneous**, i.e., all elements must have the same type. (Other programming languages may or may not have this requirement.)

For instance, this is *not* a list: `[1, "not homogeneous"]`

`list` is a **type constructor** (similar to `->` for function types, or `*` for product types). It takes a single argument, namely the type of the list elements.

Type constructors in SML are usually written in postfix notation, i.e., behind their argument(s): e.g., we write `int list`, `string list`, `int list list`. (`->` and `*`, however, are written in infix notation.)

Lists: Properties

Lists contain elements. Therefore, the list type is sometimes called a **container** data type.

Properties:

- *Variability*: the number of elements in a list is arbitrary (but finite) and only determined at run time.
- *Multiplicity*: an element may appear several times in a list.
- *Linearity*: the internal structure of a list is linear.
- *Extensionality*: two lists are equal if (and only if) they contain the same elements in the same order.

(There are other container data types, e.g., sets, queues, .)

List Expressions

Lists (just like tuples) can be constructed from expressions that need to be evaluated.

```
[18, 3+9, 5-7, size "abc", 10]
  ⟶ [18, 12, 5-7, size "abc", 10]
  ⟶ [18, 12, ~2, size "abc", 10]
  ⟶ [18, 12, ~2, 3, 10]
```

Lists in SML: Inductive Definition

Lists in SML are constructed according to the following inductive definition.

- `[]` is a list, called the **empty list**.
- If `x` is an element (called the **head**) and `xs` is a list (called the **tail**), then `x :: xs` is a list.

Nothing else is a list, i.e., all lists are constructed according to these two rules.

Lists in SML: Examples

```
> [];
val it = []: 'a list

> 1 :: it;
val it = [1]: int list

> 2 :: it;
val it = [2, 1]: int list

> 3 :: it;
val it = [3, 2, 1]: int list
```

Lists: Aggregated vs. Constructed Form

The **aggregated form**

```
[e1, ..., eN]
```

is syntactic sugar for the **constructed form**

```
e1 :: (... :: (eN :: []))
```

Both represent the same expression. For instance,

```
> [1, 2] = 1 :: 2 :: [];
val it = true: bool
```

The Type of []

What is the type of the empty list, `[]`?

Depending on the context, `[]` could be a list of integers

```
> 1 :: [];
val it = [1]: int list
```

or a list of strings

```
> ["foo"] = [];
val it = false: bool
```

or a list with elements of any other type.

Thus, `[]` is a polymorphic value of type `'a list`.

About ::

The identifier `::` behaves like a binary infix operator that is

- right-associative: e.g.,
  `1 :: 2 :: []` is `1 :: (2 :: [])`,
- of the type `'a * 'a list -> 'a list`,
- has precedence 5 (lower than $+$, but higher than $=$, $>$).

```
> [];
val it = []: 'a list
> op:: ;
val it = fn: 'a * 'a list -> 'a list
```

Pattern Matching and Recursion for Lists

Pattern Matching for Lists: Aggregated Form

Remember that data type "skeletons" can be used as patterns. Suppose `p1`, ., `pN` are patterns for the same type `T`. Then

```
[p1, ..., pN]
```

is a pattern for type `T list`. It matches a list of length `N` if `p1`, ., `pN` match the corresponding list elements.

```
> val [x, y, z] = [1, 2, 3];
val x = 1: int
val y = 2: int
val z = 3: int
> val [_, y, 3] = [1, 2, 3];
val y = 2: int
> val [x,y,z] = [1,2];
Exception- Bind raised
```

Pattern Matching for Lists: Constructed Form

Also the constructed form can be used for list patterns. Suppose `p` is a pattern for type `T`, and `ps` is a pattern for type `T list`. Then

```
p :: ps
```

is a pattern for type `T list`. It matches a list if `p` matches the head and `ps` matches the tail of the list.

```
> val x::xs = [1,2,3];
val x = 1: int
val xs = [2, 3]: int list
> val x::y::_ = [1,2,3];
val x = 1: int
val y = 2: int
> val x::xs = [];
Exception- Bind raised
```

Note: `::` is not really a function (function calls are not allowed in patterns!), but is called a (data type) **constructor**. The list data type has two constructors, namely `[]` and `::`. These *can* be used in patterns.

Lists vs. Tuples

Tuples: example `(3, 8.0, 5>8)`

- Fixed number of components
- Heterogeneous (components of possibly different types)
- Direct access to the components via `#i` selectors

Lists: example `[3, 8, 5]`

- Arbitrary length
- Homogenous (elements of the same type)
- Access via pattern matching, i.e., sequential access to the elements

Data Abstraction

By now, we have seen *all* primitive operations for lists.

- Lists are constructed using `[]` and `::`.
- Lists can be taken apart using `[]` and `::` patterns.

All other functions that construct or operate on lists are (ultimately) implemented in terms of these primitives.

This is (data) abstraction: we don't need to know how lists are actually represented in memory.

Simple List Functions: `null`, `hd`, `tl`

Using pattern matching, we can write simple functions for lists. For instance,

```
fun null  []  = true
  | null  _   = false

fun hd (x :: _) = x
  | hd _ = raise Empty

fun tl  (_ :: xs) = xs
  | tl  _ = raise Empty
```

Recursive List Functions: `length`

Using pattern matching and recursion, we can write further functions for lists. For instance,

$$\begin{aligned}&\textbf{fun}\ \text{length}\ [] = 0\\ &\quad \mid\ \text{length}\ (\_ :: \text{xs}) = 1 + \text{length xs}\end{aligned}$$

Evaluation of `length` follows the usual evaluation rules for function application (using pattern matching for lists). For instance,

```
length ["hello", "world" ^ "!"]
  ⟶ length ["hello", "world!"]
  ⟶ 1 + length ["world!"]
  ⟶ 1 + (1 + length [])
  ⟶ 1 + (1 + 0)
  ⟶ 1 + 1
  ⟶ 2
```

Structural Recursion

Many recursive functions for lists follow the same recursion scheme:

```
fun f (..., [],      ...) = ...
  | f (..., x::xs,   ...) = ... (f (..., xs, ...)) ...
```

This is known as **structural recursion** (i.e., recursion over the structure of lists). Note the similarity to simple recursion for integers.

Finding the Last Element

`last xs` returns the last element of a non-empty list `xs`: e.g., `last [1,2,3] = 3`.

Finding the Last Element

`last xs` returns the last element of a non-empty list `xs`: e.g., `last [1,2,3] = 3`.

```
fun last [x]      = x
  | last (_::xs)  = last xs
  | last _        = raise Empty
```

Note that the number of operations required to evaluate `last xs` depends on the length of `xs`. (More precisely, if the list `xs` has length $n > 0$, evaluating `last xs` performs $n - 1$ recursive calls.)

Only the *head element* of a list can be accessed easily!

Concatenating Two Lists

`append (xs, ys)` returns the concatenation of `xs` and `ys`: e.g., `append ([1], [2,3]) = [1,2,3]`.

Concatenating Two Lists

`append (xs, ys)` returns the concatenation of `xs` and `ys`: e.g., `append ([1], [2,3]) = [1,2,3]`.

```
fun append ([],  ys) = ys
  | append (x::xs,  ys) = x ::  append (xs,  ys)
```

List Reversal

`rev xs` returns a list consisting of `xs`'s elements in reverse order: e.g., `rev [1,2,3] = [3,2,1]`.

List Reversal

`rev xs` returns a list consisting of `xs`'s elements in reverse order: e.g., `rev [1,2,3] = [3,2,1]`.

```
fun rev [] = []
  | rev (x::xs) = append (rev xs, [x])
```

Note that evaluation of `rev xs` requires a number of recursive calls to `append` (depending on the length of `xs`). We will later see a better way to implement `rev`.

List Membership

`member (x, ys)` returns `true` if (and only if) `x` is an element of `ys`: e.g., `member (2, [1,3,2]) = true`, `member (2, [1,3,0]) = false`.

List Membership

`member (x, ys)` returns `true` if (and only if) `x` is an element of `ys`: e.g., `member (2, [1,3,2]) = true`, `member (2, [1,3,0]) = false`.

```
fun member (x, []) = false
  | member (x, y::ys) = x=y orelse member (x, ys)
```

Note the type of `member`: list elements must be of type `''a`, i.e., of an equality type.

```
val member = fn: ''a * ''a list -> bool
```

Aggregated vs. Constructed Form: Usage

Code should be readable and concise.

The aggregated form of lists `[x1, ..., xN]` is mostly used for .

- lists given explicitly,
- results (when displayed by Poly/ML).

The constructed form `x::xs` is mostly used for .

- decomposition of a list by pattern matching,
- construction of a list from its head and tail.

Example: Merge Sort

Merge Sort

Merge sort is a sorting algorithm. To sort a list (of length $l \geq 2$),

1. split the list into two sublists, each (roughly) of length $l/2$;
2. (recursively) sort both sublists;
3. merge the sorted sublists into a single sorted list.

Merge Sort: split

```
(* split xs
   TYPE: 'a list -> 'a list * 'a list
   PRE: true
   POST: a pair of lists (ys,zs) such that the length of ys and zs
      differs by at most 1, and ys @ zs is some permutation of xs
*)
```

Merge Sort: split

```
(* split xs
   TYPE: 'a list -> 'a list * 'a list
   PRE: true
   POST: a pair of lists (ys, zs) such that the length of ys and zs
      differs by at most 1, and ys @ zs is some permutation of xs
 *)
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (y :: z :: xs) =
      let
        val (ys, zs) = split xs
      in
        (y :: ys, z :: zs)
      end
```

Merge Sort: merge

```
(* merge (xs,ys)
   TYPE: int list * int list -> int list
   PRE: xs and ys are sorted
   POST: a sorted permutation of xs @ ys
*)
```

Merge Sort: merge

```
(* merge (xs,ys)
   TYPE: int list * int list -> int list
   PRE: xs and ys are sorted
   POST: a sorted permutation of xs @ ys
 *)
fun merge ([], ys) = ys
  | merge (xs, []) = xs
  | merge (x::xs, y::ys) =
      if x <= y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)
```

Merge Sort in SML

```
(* mergesort xs
   TYPE: int list -> int list
   PRE: true
   POST: a sorted permutation of xs
 *)
```

Merge Sort in SML

```
(* mergesort xs
   TYPE: int list -> int list
   PRE: true
   POST: a sorted permutation of xs
 *)
fun mergesort [] = []
  | mergesort [x] = [x]
  | mergesort xs =
      let
        val (ys, zs) = split xs
      in
        merge (mergesort ys, mergesort zs)
      end
```

List Operations in the SML Basis Library

Lists Operations in the SML Basis Library

The SML Basis Library already provides many of the basic list operations that we have seen in this lecture. You don't need to declare them yourself!

```
> null;
val it = fn: 'a list -> bool
> hd;
val it = fn: 'a list -> 'a
> tl;
val it = fn: 'a list -> 'a list
```

(`null`, `hd` and `tl` are actually seldomly used. Pattern matching usually results in more readable and concise code.)

Lists Operations in the SML Basis Library (cont.)

Further list functions provided by the SML Basis Library:

```
> length;
val it = fn: 'a list -> int
> List.last;
val it = fn: 'a list -> 'a
> rev;
val it = fn: 'a list -> 'a list
```

List concatenation is performed by a right-associative infix operator `@`:

```
> op @;
val it = fn: 'a list * 'a list -> 'a list
```

See `http://www.sml-family.org/Basis/list.html` for even more functions and documentation.

Lists of Characters/Strings

`explode` transforms a string into a list of characters.

```
> explode "hello";
val it = [#"h", #"e", #"l", #"l", #"o"]: char list
```

`implode` transforms a list of characters into a string.

```
> implode [#"h", #"e", #"l", #"l", #"o"];
val it = "hello": string
```

`concat` concatenates a list of strings into a string.

```
> concat ["he", "ll", "o"];
val it = "hello": string
```

Equality on Lists

In SML, equality on lists is structural, i.e., two lists are equal if and only if they contain equal elements (in the same order).

```
> [1] = [1];
```

Equality on Lists

In SML, equality on lists is structural, i.e., two lists are equal if and only if they contain equal elements (in the same order).

```
> [1] = [1];
val it = true: bool
> [1,2] = [2,1];
```

Equality on Lists

In SML, equality on lists is structural, i.e., two lists are equal if and only if they contain equal elements (in the same order).

```
> [1] = [1];
val it = true: bool
> [1,2] = [2,1];
val it = false: bool
> [1] = [1,1];
```

Equality on Lists

In SML, equality on lists is structural, i.e., two lists are equal if and only if they contain equal elements (in the same order).

```
> [1] = [1];
val it = true: bool
> [1,2] = [2,1];
val it = false: bool
> [1] = [1,1];
val it = false: bool
```

Type `'a list` is an equality type if (and only if) `'a` is an equality type.

```
> [1.0] = [1.0];
Error-Type error in function application.
...
```