More on data types

Dr. Ritwik Banerjee

Programming Abstractions

Stony Brook University, Computer Science

Scalar and Composite Types

- At the very top, types can be divided into **scalar** and **composite** types.
- A **scalar type** is a type whose values
  - occupy a fixed amount of memory, and
  - are *atomic*, that is, a value is not subdivided further in any way.
  - e.g., `int` in C, C++, and Java.
- A **composite type** is a type whose values are composed of simpler component values. That is, it is an aggregation of other simpler types.
  - Each component can be a scalar type, or itself be a composite type.
  - e.g., `int[]` in C, C++, and Java.

Primitive Types

- A **primitive type** is a type that is not defined in terms of any other type.
- All languages have some fixed set of primitive types. *E.g.*,
  - In Java: **`int`**, **`short`**, **`long`**, **`byte`**, **`char`**, **`float`**, **`double`**, **`boolean`**
- Primitive types are the basic building blocks for other data types.
- Some people consider "reference" or "pointer" to be a separate primitive type.
- All languages have the primitive types 'built-in'.
- Some languages also offer a few built-in composite data types, e.g., **`tuple`** in the ML-family and Python, and **`list`** in Java, Python, and the ML-family).
  - We do *not* consider these built-in types to be primitive.

Boolean Type

- The **boolean** data type is perhaps the simplest of all: it has exactly two values, **`true`** and **`false`**.
- The implementation of this (and other) data types is specific to a language's design and implementation.
  - Originally, C did not have a separate boolean type. The integer 0 was equivalent to **`false`**, and any other integer value was equivalent to **`true`**.
  - Since 1999, however, the ISO standard specifies an explicit boolean type in C.
  - Java supports an explicit boolean type, and enforces its use in all contexts that must evaluate to true or false (e.g., if-condition and loop-conditions). It does not perform type coercion between **`int`** and **`boolean`**.
  - Scheme is another language that supports an explicit boolean type, but it performs type coercion between **`int`** and **`boolean`**.
  - Weak dynamically typed languages like JavaScript perform many more coercions. Some of these coercions as quite scary! Feel free to take a look here: https://dorey.github.io/JavaScript-Equality-Table/

Numeric Types

Almost all languages support integer and floating-point (with various precisions) numeric types.

Most languages follow the IEEE-754 standard for the definition of *precision*:

- a single precision floating-point is defined with a 24-bit precision
- a double precision floating-point is defined with a 53-bit precision

Integers are (usually) fixed-precision as well.

- but there is no IEEE standard for this, since the universal way to represent an integer is to use the two's complement binary notation.

Since these types have fixed-precision,

- only a finite range of values can be represented.

Character and String Types

- In most languages, characters are internally represented as an integer type.
- Each integer character code represented a particular **`char`** value.
  - Depending on the encoding (ASCII, Unicode, Latin-1, etc.), a **`char`** is represented by a 8, 16, or 32-bit integer.
- Since text information is important in computation (e.g., a compiler), many languages provide **`String`** as a built-in data type, internally represented by a sequence of characters (i.e., it is not a scalar or a primitive).
- There are a few crucial issues when it comes to string representation, and the solutions are language-specific:
  a) fixed-length or dynamic-length?
  b) what operations should be supported?
  c) what data structure should be used to internally represent and/or implement it?

Character and String Types

Strings in C

- A string is represented as a `char[]`
- They can be of arbitrary length. The end of a string is denoted by `NULL` (which is '`\0`').
- There are very few built-in operations on strings.
- This is a simple representation, but since array-access is bound-checked in C, string operations are also prone to buffer overflow errors.

Character and String Types

Strings in Java

- A string is an instance of the **`java.lang.String`** class.
- Internally, this class uses a **`char[]`** to store a string.
- A string is **immutable** in Java.
  - That is, a string cannot be modified.
  - Every operation to transform a string actually produces a new string instance.
  - For example, there is no way to replace or modify a specific character in a string.

Character and String Types

Strings in OCaml

- A string is an immutable fixed-length sequence of single-byte characters.
- Each character can be accessed in constant time through its index.

  `String.get s n` or `s.[n]`, are semantically equivalent, of type `string -> int -> char`

  - Until quite recently (versions before 4.02), OCaml strings were modifiable. As such, strings and byte sequences were interchangeable.
    - This option is still available, but will soon be deprecated.
    - Just as a comparative note, OCaml strings can contain the null character (unlike C). This is because OCaml internally uses memory blocks with tags to determine a lot of things about various data types.

Ordinal Types

An **ordinal type** is a type where the values belong to a finite range of integer values.

Clearly, all integer types are ordinal types.

Characters are ordinal types.

Boolean is an ordinal type as well.

Enumeration Types

- An **enumeration** (or **enumerable**) type consists of a (typically small) finite set of values explicitly enumerated by the programmer.
    - Thus, an enumeration type is also an ordinal type.
    - It can also be considered to be a scalar *and* a primitive type.

```
/* C, C++, or Java definition of an
   enumeration type for the 12 months */
enum Month {
    JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
}
```

Subrange Type

- The **subrange type** is a contiguous sequence of an ordinal type. *E.g.*, we could define a subrange types as follows, based on a **totally ordered** data type:

```
type teen        = 13 .. 19;
type test_score = 0 .. 100;
type workday    = MONDAY .. FRIDAY;
```

  - Subrange types are defined in PASCAL and all languages derived from it.
  - They are also defined in Python, and frequently used as a **constraint** to define a subset of a data type. E.g., in an expression like

```
    for x in range(1:10):
```

Composite Types

A **composite data type** or a **compound data type** is a type that can be constructed from the language's primitive data types and other composite types.

- **Record** (or **union**)
- **Array**
- **Set**
- **List**
- **Map**
- **Pointer** (or **reference**)

Records

A **composite data type** or a **compound data type** is a type that can be constructed from the language's primitive data types and other composite types.

- An important composite data type is the record. A record can be thought of as a row in a spreadsheet, where each field in the record corresponds to a column.
- In C, this is implemented using the `struct` keyword:

```c
struct Vector {
    float x;
    float y;
    float z;
};
```

```c
struct Color {
    unsigned int red;
    unsigned int green;
    unsigned int blue;
};
```

```c
struct Vertex {
    Vector v;
    Color  c;
};
```

Records

- In most modern programming languages, the fields of these data types are accessed using **path expressions**:

```
vertex.v.x;
vertex.c.red;
```

```
struct Vector {
    float x;
    float y;
    float z;
};
```

```
struct Color {
    unsigned int red;
    unsigned int green;
    unsigned int blue;
};
```

```
struct Vertex {
    Vector v;
    Color  c;
};
```

Unions

- Some languages allow **union types**, where multiple data types share the *same* memory location. It is up to the programmer to access a union type *as if* only a specific data type exists there.
  - This was a design feature in the early years of programming, when memory was severely constrained and very expensive.
  - The size of the union is decided by its largest member (in this case, the double d). Exactly which bytes of d are considered to be i and b is implementation-dependent.
- Unions find relatively rare use nowadays, since memory is cheaper.
  - Unions are, as you can perhaps imagine, a bit tricky to work with!

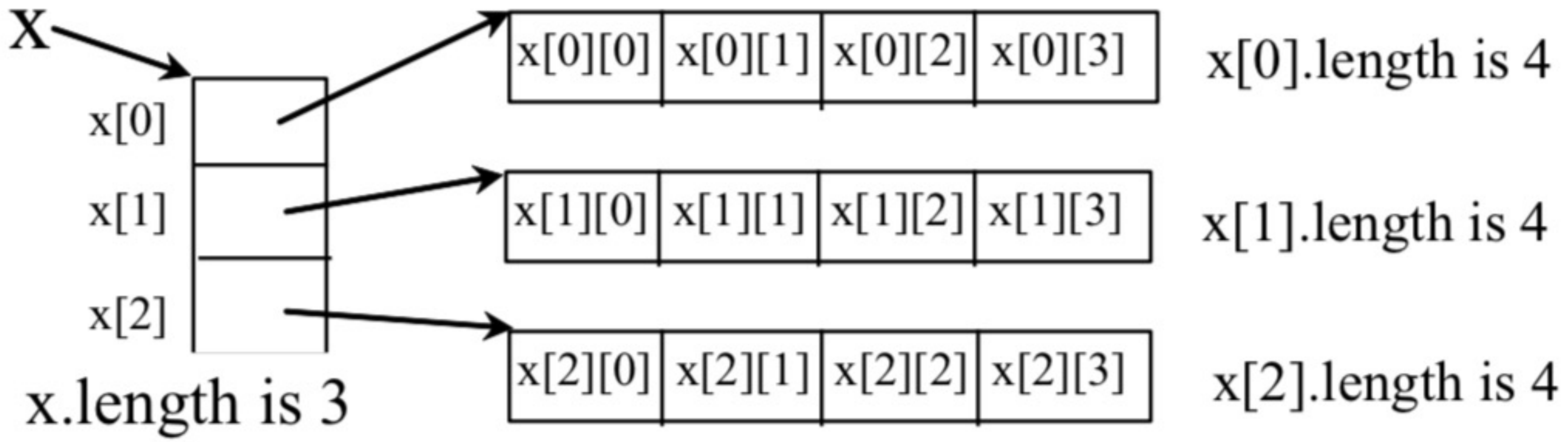
```
union {
    int i;
    double d;
    _Bool b;
}
```

Arrays

Arrays are perhaps the most widely used composite data types.

- Unlike records, which group related fields of disparate types, arrays are usually homogeneous. Semantically, they can be thought of as a mapping from an index type to a component or element type.
- Areas of memory of the same type.
- Stored consecutively, so element access for *any* index is a constant-time operation.
- Commonly, arrays are one-dimensional.
- But the data type of each array element can, once again, be an array! So we can have multi-dimensional arrays.

```
int[][] x = new int[3][4];
```


x
x[0]
x[1]
x[2]
x.length is 3
x[0][0] x[0][1] x[0][2] x[0][3]
x[1][0] x[1][1] x[1][2] x[1][3]
x[2][0] x[2][1] x[2][2] x[2][3]
x[0].length is 4
x[1].length is 4
x[2].length is 4

Arrays

Memory allocation of arrays

If the **shape** of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in **static global memory**.

If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine’s stack frame at run time, i.e., we have **stack allocation**.

Finally, if the shape is also not known at compile time, then the array is allocated memory in the heap, i.e., we have **heap allocation**.

Array indexing

Most programming languages require array indices to be non-negative integers. That is, the "index" is implicitly the integer data type.

Some languages provide **bound checking**, and raise an exception when you try to access elements outside the bounds of the array.

Other languages, like C, don't perform such checks, so the responsibility lies with the programmer.

Array operations

- Often, it is useful to get to a subsection of the array. In some programming languages, this is verbose.
- Others, like Python, offer easy **slicing-and-dicing** operations:

```
a[start:end] # items start through end-1
a[start:]    # items start through the rest
             # of the array
a[:end]      # items from the beginning all
             # the way to end-1
```

Strings and Arrays

As we saw earlier, the string data type is often internally represented using an array.

If a language allows dynamic sizing of strings, then it can’t internally use arrays (unless there are expensive array resizing operations done in the background).

Why can’t arrays *in general* be given dynamic sizing?

Because unlike strings, an array can be (i) *multi-dimensional*, and (ii) *circular*.

Sets

A **set** contains **distinct** elements without any order of traversal. Many languages offer the basic set operations: union and intersection.

A set can be implemented as a hash table, where the keys are the set elements, and there are no values associated with the keys.

Lists

A **list** is a discrete sequence of elements that allows duplicates.

Usually implemented either as

1. a linked data structure with repeating nodes, or
2. an array.

   The array-backed implementation is efficient for element access, but since arrays are not dynamically sized, there are expensive array doubling/halving operations that may happen in the background, invisible to the programmer.

Maps

A **map** is a collection of key-value pairs such that a key appears only once in the collection.

It is also called a **dictionary** or an **associative array**.

Some programming languages (*e.g.*, Lua) actually use maps to implement lists, where the list 'index' is implemented as an integer key in a map.

Maps can be implemented much like sets, using a hash table.

Recursive Types

- A **recursive type** is a data type whose objects contain one or more references to other objects of the *same* type.

- Used to build records or linked data structures like lists and trees.
  - Even intuitively, we can think of these structures as recursive. For example, an OCaml list can be defined* as

```
type 'a list = [] | :: of 'a * 'a list
```

- In both scenarios (*i.e.*, heterogeneous fields in records as well as accessing another object of the same type in linear linked data structures), the *access* to the object is carried out efficiently – and sometimes quite intuitively – by **pointers**, also known as **references**.

* This is NOT how it's actually defined in OCaml.

Pointers and Addresses

Pointers and addresses are closely related, but the two are *not* the same!

A pointer is a high-level abstract concept – it's a reference to an object.

The address is a low-level concept tied to machine architecture – it's the location of a token in memory.

Pointers are usually *implemented as* addresses (but not necessarily).

Pointers, arrays, and pointer arithmetic

- Pointers allow the use of **pointer arithmetic** – given a pointer to an element of an array, the addition of an integer *k* produces a pointer to the element *k* positions later.

```
char s[] = “hello world”;  /* create a char array */
char* p = s;               /* the array name is a pointer to its 1st item, so
                              create an alias. now, p and s are the same object. */
for ( ; *p != ‘\0’; p++) { /* no initialization in the for loop; the loop-condition
    cout << *p;               checks if the end of the char array is reached; the
}                             iteration is done using pointer arithmetic: p++ */
```

  - A negative value of *k* is valid, and will produce a pointer to the element *k* elements earlier.
  - Pointer arithmetic is only useful in arrays. So that makes it a rather implementation-oriented *un*-abstract way of coding.
  - Pointer arithmetic is most commonly used by C programmers, but C does not check for array bounds. As a result, pointer arithmetic is prone to buffer overflows.

Pointers, arrays, and pointer arithmetic

```
1.  int n;       /* declare an integer */
2.  int *a;      /* use the deref operator * to create a pointer to int */
3.  int b[10];   /* declare an array of 10 integers */
4.
5.  /* Now, the following are all valid C statements: */
6.  a = b;       /* create an alias */
7.  n = a[3];    /* n is the 4th item in the array */
8.  n = *(a+3);  /* equivalent to the previous line */
9.  n = b[3];    /* equivalent to lines 7 and 8 */
10. n = *(b+3);  /* equivalent to lines 7, 8, and 9 */
```

- We may not like pointer arithmetic, but the syntax used in C arrays, `[]`, is actually just syntactic sugar over pointer arithmetic!
- The expression `A[b]` is defined to be `(*((A)+(b)))`, which is the same as `(*((b)+(A)))`.
  - Something most programmers don't realize: `A[3]` is actually equivalent to `3[A]`.

Java References versus C/C++ Pointers

- Languages like Java support **references**, which act a lot like pointers in C or C++. But there are some really important differences:
  1. Java references are not allowed pointer arithmetic. We can only *de*reference it, or compare it with another reference.
  2. Java references cannot be cast into an incompatible type. This is because Java has strong static typing.

Since we are not going to get into the details of C or C++ in this course, we will use the terms *pointer* and *reference* interchangeably (even though ‘reference’ is the technically correct term to be used in the context of Java).

Uninterpreted types

- As we have seen, every language provides some **base** (or **atomic**) types.
  - Consist of sets of simple, unstructured values (e.g., integer, boolean), together with a set of operations to manipulate these values.
- To understand the overall theory of data types, it is also useful to think that a language comes with a set of **uninterpreted** (or **unknown**) types, with no corresponding operations at all.
  - In an abstract sense (e.g., in λ-calculus), we can think of the set of uninterpreted types as a *placeholder* for the base types.
  - When we speak of a particular programming language, we will usually no longer need this abstraction because we can directly deal with the base types of *that* language.
  - But … uninterpreted types are not useless! Even though we cannot name its elements directly, we can still bind variables. For example:

```
(* identity function on the elements of an unknown base type *)
fun: 'a -> 'a

(* repeat g : 'a -> 'a twice, where the argument’s type is unknown *)
fun: (g x) -> g (g x)
```

Algebraic data types

- Programs often need to deal with collections of values (often heterogeneous).
  - A tree is either empty, or a node with children where each child is, again, a tree.
  - A list is either **`nil`** (i.e., `[]`) or a cons (i.e., `::`) with a head and a tail.
- The theoretical mechanism that supports this is called algebraic data types.
- An **algebraic data type** (or, a **variant type**) is a data type representing a value that is one of several possibilities, the simplest variant types we have seen are ordinals, e.g., the **`enum`** type in Java.
- We can declare such types in OCaml as well:

```
# type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat;;
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
# let d:day = Tue;;
val d : day = Tue
```

Algebraic data types

- As illustrated by the example, the syntax to define a variant type is
  `type t =` $\mathtt{C_1 \mid C_2 \mid \ldots \mid C_n}$
  - The constant values $\mathtt{C_i}$ are called *constructors* in OCaml (different from the notion of constructors in Java), and they must start with an uppercase letter.
  - A constant is already a fixed value, so no additional runtime semantics are needed.

- Consider two type definitions that seem to be overlapping:

```
# type t1 = C | D;;
# type t2 = D | E;;
# let x = D;;
```

  What is the type of **`x`** here?
  - It takes the type defined later, i.e., **`t2`**.
  - Even though such overlapping types are 'allowed', it is *never* recommended that you code this way!

Pattern matching with variants

- Unlike the **`enum`** type in Java, OCaml does not have any intrinsic mapping of a constant to a corresponding **`int`** value. Pattern matching must be explicit to create such a mapping:

```
let int_of_day d = match d with
    | Sun -> 1
    | Mon -> 2
    | Tue -> 3
    | Wed -> 4
    | Thu -> 5
    | Fri -> 6
    | Sat -> 7;;
```

Pattern matching with variants

Algebraic data types can do a lot more than just enumerate. As an example, we can define a **shape** type and a few functions that use it:

```
# type point = float * float;; (* a two-dimensional point *)
# type shape =
    | Point of point
    | Circle of point * float (* center and radius *)
    | Rect of point * point (* bottom-left and upper-right corners *);;
# let area = function
    | Point _ -> 0.0
    | Circle (_,r) -> pi *. (r ** 2.0)
    | Rect ((x1,y1),(x2,y2)) ->
        let w = x2 -. x1 in
        let h = y2 -. y1 in w *. h;;
# let center = function
    | Point p -> p
    | Circle (p,_) -> p
    | Rect ((x1,y1),(x2,y2)) -> ((x2 -. x1) /. 2.0, (y2 -. y1) /. 2.0);;
```

Non-constant values

- The **`shape`** data type is a variant type that is defined in terms of a collection of constants (i.e., ‘constructors’ in OCaml).
- What is different here, compared to the simple enumeration types, is that each constructor may carry additional data.
  - Those that carry additional data are *non-constants*.
- Every instance of an algebraic data type is formed from exactly one of these constructors. This is sometimes called “tagging”.
  - For the **`shape`** type, every instance is tagged as **`Point`**, **`Circle`**, or **`Rect`**.

Type definitions

- Types are defined with the reserved keyword **`type`**, which is recursive by default (unlike **`let`**). It has global scope.

```
type name = type-definition;;
```

- The **`and`** keyword can be used to define mutually recursive types:

```
type
    name₁ = type-definition₁
and name₂ = type-definition₂
     ⋮
and nameₖ = type-definitionₖ;;
```

- This is OCaml's way of defining a record type. E.g.,

C	`struct intpair { int a, b; };`
OCaml	`type intpair = { a : int; b : int };;`

- Creating such objects is easy:

```
# {a=3; b=5};;
- : intpair = {a = 3; b = 5}
```

Sum and product types

- These are the simplest variant types that move beyond just enumeration.
  - The **sum type** describes a variant whose value is drawn from *exactly one of the constructors*. The **`day`** type was an example of a sum type.
  - The **product type** describes a variant whose value is drawn from a constructor that can carry *tuples* or *records*, and the values in these tuples or records have a sub-value from each of their component types.
  - For example, here is a simple product type whose value is based on a record-type constructor, and the record has sub-values of the type **`string`**.

```
# type name = {first : string; last : string};;
type name = { first : string; last : string; }
# let johndoe = {first = "john"; last = "doe"};;
val johndoe : name = {first = "john"; last = "doe"}
# johndoe.first;;
- : string = "john”
```

? *Was the* `intpair` *data type a sum type, or a product type?*

Unions

- Recall the unsafe union type where multiple objects share the same memory.
- Most modern languages don't allow it, but older languages like C do. This is what a union definition would look like in C*:

```
struct foo {
    int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
  union {
    int i;        // If type == TYPE_INT.
    int pair[2]; // If type == TYPE_PAIR_OF_INTS.
    char *str;   // If type == TYPE_STRING.
  } u;
};
```

* Source: https://ocaml.org/learn/tutorials/data_types_and_matching.html#Variants-qualified-unions-and-enums

Unions

- The equivalent in OCaml is just the record type we have seen before.
  - It is 'safe' to access, unlike the union type in C.
  - The definition is much more concise.
  - Uses the same pattern matching template.

```
# type foo =
     | Nothing
     | Int of int
     | Pair of int * int
     | String of string;;
type foo = Nothing | Int of int | Pair of int *
int | String of string
```

* Source: https://ocaml.org/learn/tutorials/data_types_and_matching.html#Variants-qualified-unions-and-enums

Unions

- Using such union types, we can create *heterogeneous* lists where, say, some items are strings and the others are integers.

```
# type str_or_int_list = str_or_int list;;
type str_or_int_list = str_or_int list
# let rec sum : str_or_int list -> int = function
    | [] -> 0
    | (String s)::t -> int_of_string s + sum t
    | (Int i)::t -> i + sum t;;
val sum : str_or_int list -> int = <fun>
# sum [String "11"; Int 4; String "5"];;
- : int = 20
```

Unions

- With algebraic data types, the previous examples show that we can distinguish between the subtypes using the constructor.
  - If an element is of type **`str_or_int`**, we can trace the definition and check whether it is a **`string`** or an **`int`**.
- This feature can be extremely helpful in tree data types, where the recursive structure has multiple branches:

```
# type t = Left of int | Right of int;;
type t = Left of int | Right of int
# let x = Left 10;;
val x : t = Left 10
# let y = Right 10;;
val y : t = Right 10
# let increment_right = function
  | Left i -> i
  | Right i -> i+1;;
val increment_right : t -> int = <fun>
# increment_right x;;
: int = 10
# increment_right y;;
: int = 11
```

Recursion with algebraic data types

```
# type intlist = Nil | Cons of int * intlist;;
type intlist = Nil | Cons of int * intlist
# (* like 3::[ ] *)
  let lst1 = Cons(3,Nil);;
val lst1 : intlist = Cons (3, Nil)
# (* like 1::2::3::[ ], which is just [1;2;3] *)
  let lst = Cons(1, Cons(2, Cons(3, Nil)));;
val lst : intlist = Cons(1, Cons (2, Cons (3, Nil)))
# let rec length:intlist -> int = function
   | Nil -> 0
   | Cons(_,xs) -> 1 + (length xs);;
val length : intlist -> int = <fun>
# length lst;;
-  : int = 3
# (* records are algebraic; recursion is allowed *)
# type person = {name: string; child_of: person};;
type person = { name : string; child_of : person; }
# (* simple type synonyms can't be recursive *)
  type t = t * t;;
Error: The type abbreviation t is cyclic
```

Defining variants with the uninterpreted type

- Note that the list we defined using **`Cons`** need not be specific to the **`int`** type. As such, we should be able to generalize a list across *any* possible data type.
- In other words, we want to make our list **generic** (i.e., **polymorphic**), and for that we can use the uninterpreted type in OCaml, using the **`'a`** notation:

```
# type 'a mylist = Nil | Cons of 'a * 'a mylist;;
type 'a mylist = Nil | Cons of 'a * 'a mylist
# let rec length : 'a mylist -> int = function
    | Nil -> 0
    | Cons(_, xs) -> 1 + (length xs);;
val length : 'a mylist -> int = <fun>
# let lst = Cons("this", Cons("is", Cons("my",
 Cons("list", Nil))));;
val lst : string mylist = Cons ("this", Cons
 ("is", Cons ("my", Cons ("list", Nil))))
# length lst;;
- : int = 4
```