Recursive Data Types

Luca Abeni

December 14, 2022

1 Data Types

Various programming languages allow users to define new data types by defining the set of valid values and the operations that can be performed on these values.

An example of user-defined data types (perhaps the simplest case of a user-defined type) is represented by enumerative data types. For example, you can define a "`color`" type with values "`red`", "`blue`", and "`green`" and then define the various operations you can perform on those values. This way of defining new data types is clearly not practical when the number of values is very large (and does not allow defining new data types when the number of values is not finite). To address this issue, the concept of enumerative type can be generalized by using *data constructors* instead of simple constant values like "`red`", "`blue`", and "`green`". The new data type that is going to be defined therefore has values generated by one or more constructors, which operate on one or more arguments. It is clear that a constructor having an argument of integer type can generate a large number of values. Also note that the "`red`", "`blue`", and "`green`" values mentioned above are nothing more than special cases of data constructors[1].

The values of the new data type are then partitioned into various subsets, called *variants*; in other words, the set of possible values (definition set of the data type) is the disjoint union of the variants of the data type. Each variant is associated with a constructor, which generates all the values composing the variant; in other words, a variant is the set of values generated by a single data constructor. Note that if we assume that two different constructors can never generate the same value, we can say that the definition set of the data type is the union (or sum between sets) of the variants (instead of "disjoint union"). It will become clear later on why the concept of "sum" is interesting in this context.

Using the Standard ML syntax, it is possible to define a new data type in a very generic way with

```
datatype <name> = <cons1> of [type1] | <cons2> of [type2] | ... ;
```

where the "`<cons1>`", ... "`<consn>`" constructors represent the type variants. A similar definition in Haskell would be

```
data <name> = <cons1> [type1] | <cons2> [type2] | ... | <consn> [typen]
```

(notice that for Haskell the data type names and the constructor names must start with a capital letter). Note that constructors are real functions that take an argument and map it into a value of the new data type. In some languages like Standard ML, if you need multiple arguments for a constructor you can use a tuple argument; in other languages like Haskell a constructor can return a function that returns the value of the new data type (or a function again...), using the currying technique. Haskell provides a simplified syntax for these situations, however, which makes valid definitions like

```
data Test = Constructor1 Int Double | Constructor2 Char String
```

or similar. In this case, "`Constructor1`" is not a function with two parameters (as a naive reader might think), but a function accepting one single parameter (of type "`Int`") and returning a function from "`Double`" to "`Test`". "`Constructor1`" has hence type "`Int -> Double -> Test`" (which means "`Int -> (Double -> Test)`").

The example in Figure 1 shows how to define a "`Num`" data type which can have integer or floating point values.

[1]In particular, they are constructors that take no input arguments - or have 0 arguments. Thus, each constructor generates a unique value, which is identified with the name of the constructor.

```
    data Num = Integernum Int  | Realnum Double
    sumnumbers (Integernum a, Integernum b) = Integernum (a + b)
    sumnumbers (Integernum a, Realnum    b) = Realnum
(fromIntegral a + b)
    sumnumbers (Realnum    a, Integernum b) = Realnum
(a + fromIntegral b)
    sumnumbers (Realnum    a, Realnum    b) = Realnum    (a + b);
     ...
```

Figure 1: Example of data type with two constructors (receiving arguments of different types), in Haskell.

As previously mentioned, a constructor receiving more than one argument as input can use a tuple to group its arguments, thus realizing a *Cartesian product* among the argument definition sets[2]. New data types can then be created from existing data types using (disjoint) unions and Cartesian products. More simply, we can speak of *sums or products* between existing data types.

In this way, an algebra, or algebraic structure, of data types is defined, consisting of a support set - the set of data types - with a sum operation and a product operation defined on it. The name "*Algebraic Data Types*" (ADT) is hence often used to indicate the data types defined in this way.

2 Recursion

The *recursion* technique (closely related to the mathematical concept of *induction*) is used in computer science to define some kind of "entity"[3] based on itself. The typical example is represented by recursive functions: a function $f()$ can be defined by expressing the value of $f(n)$ as a function of other values computed by $f()$ (typically, $f(n-1)$). But a similar idea can also be used to define a set by describing its elements based on other elements contained in the set (example: if the element n belongs to the set, then $f(n)$ also belongs to the set).

In general, recursive definitions are given "by case", ie they are composed of several clauses. One of these is the so-called *basis* (also called *inductive basis*); then there are one or more clauses or *inductive steps* which allow to generate/calculate new elements starting from existing elements. The basis is a clause of the recursive definition that does not refer to the "entity" being defined (for example: "the factorial of 0 is 1", or "0 is a natural number", or "1 is a prime number", etc...) and has the task of ending the recursion. Without an inductive basis, an endless recursion happens (and this is not useful from a practical point of view).

It is known how it is possible to use recursion to define functions (and how recursion is the only way to implement loops in functional programming languages - in other words, recursion can be considered as a sort of "functional equivalent" of iteration). In essence, a function $f : \mathcal{N} \to \mathcal{X}$ can be defined by defining a function $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, a value $f(0) = a$ and imposing that $f(n+1) = g(n, f(n))$.

More in details, a function can be defined by recursion when its domain is the set of natural numbers (or a countable set); the codomain can instead be a generic set $\mathcal{X}$. As an inductive basis, we define the value of the function for the smallest value belonging to the domain (for example, $f(0) = a$, with $a \in \mathcal{X}$) and as an inductive step the value of $f(n+1)$ is defined based on $f(n)$. As mentioned above, this can be done by defining $f(n+1) = g(n, f(n)))$. Note that the domain of $g()$ is the set of pairs of elements taken from the domain and codomain of $f()$, while the codomain of $g()$ is equal to the codomain of $f()$.

It is important to note that the mathematical concept of induction can be used to define not only functions but also sets, properties of numbers, etc... For example, a set can be defined by induction by indicating one or more elements that belong to it (inductive basis) and by defining new elements of the set starting from elements belonging to it (inductive step). Using this technique the set of natural numbers can be defined according to the *Peano axioms*:

- Inductive base: 0 is a natural number ($0 \in \mathcal{N}$)

[2]In some languages, such as Standard ML, a data constructor is restricted to having at most one argument, so the use of the Cartesian product is mandatory. Other languages, such as Haskell, provide a syntax to create multi-argument constructors. In this second case the Cartesian product is hidden by the syntax of the language, but conceptually it is always present.

[3]The term "entity" here is used informally to generically indicate functions, sets, values... But, as we will see shortly, also data types!

```
i2n 0 = Zero
i2n x = Succ (i2n (x - 1))

n2i Zero      = 0
n2i (Succ n) = 1 + n2i n

nsum Zero      n = n
nsum (Succ m) n = Succ (nsum m n)
```

Figure 2: Various functions working on Peano encoding of the natural numbers (defined using a recursive data type).

- Inductive step: $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$ (that is, it is possible to define a function $s : \mathcal{N} \to \mathcal{N} - \{0\}$ computing the successor of a natural number)

3 Recursive Data Types

As mentioned, to define a data type it is necessary to define its definition set (set of values belonging to the data), plus the operations that can be applied to the values of the type. Since a set can be defined by induction (as seen), one can think of applying recursion to define new data types.

Recall that the values of a data type can be generated by constructors that take one or more arguments as input (actually, a n-tuple of arguments), it becomes clear how to apply recursion to types data: a constructor for a data type `T` can have an argument of type `T` (or, a tuple containing an element of type `T`). Clearly, an inductive basis is needed to avoid infinite recursion; this means that one of the variants of the type must have a constructor that has no arguments of type `T`. As an inductive step, there may be other variants with constructors that have arguments of type `T`.

For example, it is possible to define the type `natural` based on the recursive definition of the set of naturals given by Peano: the type will consist of two variants, one of which (the inductive basis) will have a constant constructor `zero` (representing the number 0), while the other will have a `succ` constructor, which generates a natural starting from a natural. Using Haskell's `data` construct, this would be

```
data Natural = Zero | Succ Natural
```

To complete the definition of the "`Natural`" data type, some simple operations of the values of this type must be defined. For example, Figure 2 shows how to implement a conversion from "`Int`" to "`Natural`" ("`i2n`" function), a conversion from "`Natural`" to "`Int`" ("`n2i`" function), and the sum of two natural numbers ("`nsum`" function).

Curious readers can check a possible C++ implementation of the Peano coding of natural numbers, shown in Figure 3. From this simple example (the implementation of the "`i2n`" function and the "`+`" operator are missing, but they are quite simple) one can immediately observe an interesting thing: since in C++ the data constructor associated with a class has the same name as the class, to have a "`zero()`" constructor and a "`succ()`" constructor it was necessary to derive two "`zero`" and "`succ`" classes from the "`natural`" abstract class. This suggests that a data type with multiple variants can be implemented in C++ by using an abstract base class representing the data type, from which a class for each variant / constructor is derived.

Another interesting thing to note is the usage of *references* to account for the recursive nature of the "`natural`" data type. This begins to suggest that there is a relationship (which will become clear as we look at lists) between recursive datatypes and references / pointers: languages that do not explicitly support recursion on datatypes are forced to introduce the pointer or reference data type, while languages that don't support pointers / references use recursive data types to avoid losing expressive power.

Finally, the "`const`" keyword is used to guarantee the immutability of the data structures encoding the natural numbers.

Although interesting from a mathematical point of view, the definition of this "`Natural`" data type is not too useful... However, there are a number of data structures that can be defined as recursive types and they are very useful: they are of lists, trees, and similar dynamic data structures. For example, a list of integers is recursively definable as follows: a list is empty or it is the concatenation of an integer and a list. The "empty list" constructor represents the inductive basis, while the "integer and list concatenation" constructor represents the inductive step. Note that a list is a recursive data structure

```
class natural {
  public:
    virtual unsigned int n2i(void) const = 0;
};

class zero: public natural {
  public:
    zero(void) {
    };
    virtual unsigned int n2i(void) const
    {
      return 0;
    };
};

class succ: public natural {
    const natural &prev ;
  public:
    succ(const natural &p) : prev(p)
    {
    };
    virtual unsigned int n2i(void) const
    {
      return prev.n2i() + 1;
    };
};
```

Figure 3: Example showing a possible C++ implementation of the Peano encoding of natural numbers.

because the "integer and list concatenation" constructor takes values of type list as its second argument. Using Haskell syntax, a list of integers can therefore be defined as

```
data List = Empty | Cons Int List
```

where `Empty` is the constructor of an empty list, while `Cons` generates the variant containing all non-empty lists. Note that the `Cons` constructor has an argument of type `Int` and returns a function `List -> List`, because a Haskell function can't have more than one argument (thus, currying is used to support multiple arguments). However, "`Cons Int List`" could also be seen as a function that takes two arguments: an integer and a list.

4 Immutable and Mutable Lists

The "`List`" recursive data type introduced in the previous section represents a very special type of list, called an "immutable list". The reason for this name can be understood by considering the operations implemented on this type. Since immutable lists (and immutable data structures in general) are mainly used in functional languages, the Haskell language will be used in the next examples (but the examples are easily translatable into other functional languages, such as those of the ML family).

The first two operations on the "`List`" data type are the two constructors `Empty` and `Cons`, which respectively generate empty lists and non-empty lists (that is, concatenations of integers and lists). To operate on immutable lists in a generic way, two other operations traditionally called "`car`" and "`cdr`" are needed. Given a non-empty list, "`car`" returns the first integer of the list (the so-called "head of the list"), while "`cdr`" returns the list obtained by removing the first element. Both "`car`" and "`cdr`" are undefined for empty lists. A simple implementation of these two functions is given by:

```
car (Cons v _) = v
cdr (Cons _ l) = l
```

Note that these definitions are not exhaustive: both "`car`" and "`cdr`" are in fact defined using pattern matching on a value of the "`List`" type, but the only pattern present in the definition matches non-empty

```
isempty Empty = True
isempty _     = False

list_len Empty      = 0
list_len (Cons _ l) = 1 + list_len l

concat Empty b      = b
concat (Cons e l) b = Cons e (concat l b)
```

Figure 4: Example of Haskell functions working o immutable lists, using only `Empty`, `Cons`, `car`, and `cdr`.

```
ins = \n -> \l ->
      if ((l == Empty) || (n < car l))
        then
          Cons n l
        else
          Cons (car l) (ins n (cdr l))
```

Figure 5: Ordered insert into a list, implemented in Haskell.

lists ("`Cons`" constructor)... There is no pattern that matches empty lists ("`Empty`" constructor). This means that if "`chr`" or "`cdr`" is applied to an empty list, an exception is thrown (not really "purely functional"). This all reflects the fact that "`car`" and "`cdr`" are not defined for empty lists.

Every operation on lists can be implemented based on `Empty`, `Cons`, `car` and `cdr` only. For instance, Figure 4 shows how to check if a list is empty, compute its length, or concatenate two lists using only the primitives just mentioned. Note that many of the functions that act on lists are recursive (since lists are defined as a recursive data type, this shouldn't be too surprising).

The "`isempty`" function is very simple and returns a boolean value based on pattern matching: if the list passed as an argument matches a list generated by the `Empty` constructor (that is, if it is a empty list), returns `True` otherwise (wildcard pattern) returns `False`.

The "`list_len`" function works recursively, using the "`Empty`" pattern as an inductive basis (the length of an empty list is 0) and saying that if the length of a list `l` is n, then the length of the list obtained by inserting any integer in front of `l` is $n + 1$ (inductive step).

Finally, the `concat` function uses the concatenation of an empty list with a generic list "`b`" as an inductive basis (by concatenating an empty list with a list "`b`", we get `b`). The inductive step is based on the fact that concatenating a list composed of an integer `n` and a list `l` with a list `b` is equivalent to creating a list composed of the integer `n` followed by the concatenation of `l` and `b`.

Finally, a function to insert an integer into a sorted list can be implemented as shown in Figure 5. When we want to insert a new element in an empty list (`l = Empty`), the resulting list is created (via `Cons`) by adding the new element to the head of the list. This is also done if the element to be inserted is smaller than the first element of the list (`n < car l`). Otherwise, a new list is created (again via `Cons`) by concatenating the head of `l` (`car l`) with the list created by inserting the number into the remainder of `l` (`insert n (cdr l)`). In any case, a list containing `n` inserted in the right position is created by one or more invocations of `Cons`, and not by modifying the list `l`.

For example, consider what happens when you invoke

```
ins 5 (Cons 2 (Cons 4 (Cons 7 Empty)))
```

since $5 > 2$, this "`ins`" expression evaluates to

```
Cons 2 (ins 5 (Cons 4 (Cons 7 Empty)))
```

and

```
Cons 2 (Cons 4 (ins 5 (Cons 7 Empty)))
```

which finally stops the recursion evaluating to "Cons 2 (Cons 4 (Cons 5 (Cons 7 Empty)))". As a result, 3 new values of type "`List`" have been created.

To better understand the behavior of an immutable list and the reason for its name, it is useful to see how it can be defined in a language that does not directly support recursive data types, such as the

```
struct list {
  int val;
  struct list *next;
};

struct list *empty(void)
{
  return NULL;
}

struct list *cons(int v, struct list *l)
{
  struct list *res;

  res = malloc(sizeof(struct list));
  res->val = v;
  res->next = l;

  return res;
}

int car(struct list *l)
{
  return l->val;
}

struct list *cdr(struct list *l)
{
  return l->next;
}
```

Figure 6: Example of immutable lists in C.

C language. In this case, pointers are used to connect the various elements of the list: in particular, each element of the list is represented by a structure composed of an integer field (the value of that element) and a pointer to the next element of the list. So, instead of having a "`list`" data type defined based on itself we have a "`list`" data type defined using a pointer to "`list`". The two "`empty()`" and "`cons()`" constructors are implemented as functions that return a pointer to "`list`" (these functions therefore dynamically allocate the memory necessary to contain a "`list`" structure) and the "`car()`" and "`cdr`" functions simply return the values of the fields of the "`list`" structure. A list is terminated by an element which has a pointer to the next element equal to `NULL` (this can be considered the equivalent of the inductive basis). In summary, a simple C implementation might look like Figure 6. Note that to simplify the implementation the result of `malloc()` is not checked (a more "robust" implementation should instead check that `malloc()` does not return `NULL`).

The "`insert()`" function can then be defined in the same way as in Haskell, using only the "`empty()`", "`cons()`", "`car()`" and "`cdr()`" functions as shown in Figure 7. This definition uses the *arithmetic if* operator, which is equivalent to Haskell's `if...then...else...end` expression but is perhaps less readable compared to a "traditional" selection construct. The implementation of Figure 8 is equivalent (albeit unstructured and "less purely functional").

Now, consider what happens when invoking l = ins(5, l); if `l` is a pointer to the list in Figure 9, containing the values 2, 4, and 7. Since 5 > 2, `ins()` recursively invokes ins(5, cdr(l));, with `cdr(l)` being a pointer to the second element of the "`l`" list (it is therefore a pointer to a list containing the values 4 and 7). The result of this recursive invocation of `ins()` will then be passed to `cons()` (which will dynamically allocate a new structure of type "`list`"). Since `car(cdr(l))` returns 4 and 5 > 4, ins(5, cdr(l)); will again recursively invoke `ins()`, with parameters 5 and `cdr(cdr(l))`. Now, since `car(cdr(cdr(l)))` returns 7 and 5 < 7, ins(5, cdr(cdr(l))); will return `cons(5, cdr(cdr(l)))`, dynamically allocating a structure of type "`list`". `cons()` will then be invoked 2 more times, and as a

```
struct list *ins(int n, struct list *l)
{
  return ((l == empty()) || (n < car(l))) ?
                        cons(n, l) : cons(car(l), ins (n, cdr(l)));
}
```

Figure 7: Ordered insert into an immutable list, implemented in C.

```
struct list *ins(int n, struct list *l)
{
  if ((l == empty()) || (n < car(l))) {
    return cons(n, l);
  } else {
    return cons(car(l), ins(n, cdr(l)));
  }
}
```

Figure 8: Ordered insert into an immutable list, implemented in C without arithmetic if.

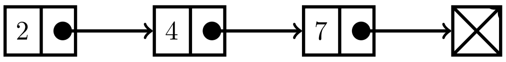

Figure 9: Example of list containing the integers 2, 4 e 7.

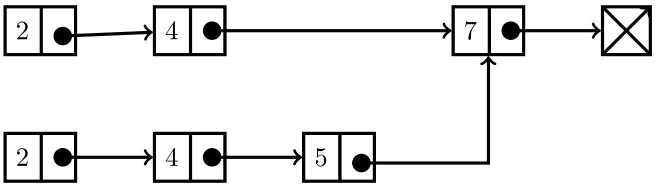

Figure 10: List of Figure 9 after ins(5, l);.

result `ins(5, l);` will return a list consisting of 3 new dynamically allocated elements (`cons()` has been called 3 times), without modifying any element of the list of Figure 9. This is why this kind of list is called "immutable": the `val` and `next` fields of the `list` structure are set when the structure is allocated (from `cons()`) and are never modified. The final result is visible in Figure 10, which shows below the 3 new dynamically allocated elements (note that now the pointer `l` points to the element of value 2 on the bottom).

It is also important to note that in the previous example any reference to the first element of `l` may have been "lost", leaving memory areas that are dynamically allocated but no longer reachable: if a program invokes

```
l = vuota();
l = ins(4, l);
l = ins(2, l);
l = ins(7, l);
l = ins(5, l);
```

the data structures that have been dynamically allocated by the second invocation of `ins()` (that is, `ins(2, l);`) no longer have any pointers "reaching" them.

Returning to the previous example, when `l = ins(5, l)` is invoked, the structures containing the first two elements of the "old list" may therefore no longer be reachable, but the memory in which they are stored has not been released by any call to `free()`. This clearly indicates that this implementation of immutable lists is likely to generate *memory leaks*, so some kind of garbage collection mechanism is needed.

The previous implementation of immutable lists in C can be compared with a "more traditional" implementation based on mutable data structures, shown in Figure 11.

```
struct list {
  int val;
  struct list *next;
};

struct list *ins(int n, struct list *l)
{
  struct list *res, *prev, *new;

  new = malloc(sizeof(struct list));
  new->val = n;

  res = l;
  prev = NULL;
  while ((l != NULL) && (l->val < n)) {
    prev = l;
    l = l->next;
  }
  new->next = l;
  if (prev) {
    prev->next = new;
  } else {
    res = new;
  }

  return res;
}
```

Figure 11: Traditional implementation of linked lists in C, without using immutable data structures.

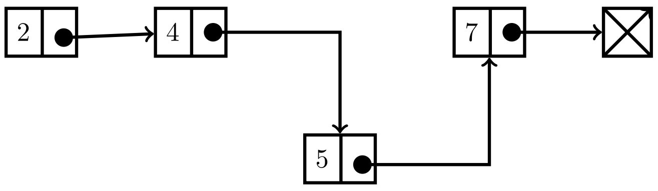

Figure 12: List of Figure 9 after ins(5, l); using “traditional” mutable lists.

Note that this implementation of the "`ins()`" function modifies the `next` field of the previous element to the one being inserted, so the list is not immutable. On the other hand, it doesn't suffer from the memory leak problems highlighted for the previous implementation (essentially, this tells us that garbage collection techniques are only necessary if immutable data structures are used). The result of `l = insert(5, l);` using this implementation of lists is visible in Figure 12 (note that the arrow going out of element "4" has changed).

```
public class List {
    private final Integer val;
    private final List next;

    public List(){
        val=null;
        next=null;
    }
    public List(int v, List l) {
        val = v;
        next = l;
    }
    public int car() {
        return val;
    }
    public List cdr() {
        return next;
    }


    public void printList() {
        if (next!=null) {
            System.out.println(val);
            next.printList();
        }
    }

    public List ins(int n){
        if (next==null || n < val)
        return new List(n,this);
        else return new List(car(),cdr().ins(n));
    }

    public static void main(String a[]) {
        List l = new List();
        l = l.ins(1);
        l = l.ins(5);
        l = l.ins(3);
        l = l.ins(9);
        l = l.ins(7);
        l.printList();
    }
}
```

Figure 13: Java implementation of immutable lists.

Finally, for the sake of a comparison Figure 13 shows an implementation of an immutable list in Java (remember that a Java VM implements a garbage collector by default, so memory leaks due to the use of immutable data structures are not a problem).

It is interesting to note that in Java the `final` qualifier allows you to explicitly specify that the fields of the class will never be modified (the data structure is immutable). Also, note that Java (like many other object-oriented languages) forces you to use the same name for constructors and for the class. Therefore, there will not be two “`Empty`” and “`Cons`” constructors for the two variants, but two constructors both called “`List`” which are distinguished by the number and type of their arguments (one constructor - that of empty list - has no arguments, while the other - the one corresponding to `Cons` - has one argument of type `int` and one of type `List`).