Recursive Data Types

Luca Abeni
luca.abeni@santannapisa.it

Recursive Data Types

- To define a data type, we must (also) define all its possible values
- Set of possible values $\rightarrow$ can be defined by induction...
- Can induction/recursion be used to define a new data type?
  - How? We need induction base and induction step
  - Induction base: one (or more) constructor(s) having 0 parameters (or, no parameters of the data type we are defining)
  - Induction step: constructor having a parameter of the type we are defining
- Looks... Confusing??? Let's look at some examples!

Recursive Data Types: Example

- Let's define the "natural numbers" data type (set of values: $\mathcal{N}$)
  - $0 \in \mathcal{N}$: constructor `zero` (with no parameters)
  - $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$: constructor `succ`, having as an argument a natural number

```
datatype nat = zero | succ of nat;
```

```
data Nat = Zero | Succ Nat
```

- How to use this funny definition?
  - Combination of *pattern matching* and *recursion*
  - Familiar to people knowing functional programming

More Interesting Example: Lists

- Lists can also be defined by induction/recursion (simple example: list of intergers)
  - Inductive base: an empty list is a list
  - Inductive step: A non-empty list is an integer followed by a list
- Recursive Data Type: a non-empty list is defined based on the list data type (constructor receiving a list as a parameter)
- Two constructors
  - Empty list constructor
  - Constructor for non-empty lists

Lists as RDTs — 1

- Two constructors
  - Empty list constructor (no parameters)
  - Constructor for non-empty lists (two parameters: an integer and a list)
- Other operations
  - `car`: returns the first element of a non-empty list (head)
  - `cdr`: given a non-empty list, returns the “rest of the list”

Lists as RDTs — 2

- How are lists generally implemented?
- Functional languages (Haskell, ML Lisp & friends, ...)
  - Recursive data type!!!
  - "`cons`" constructor: parameter of type `int * list` (or, a parameter of type `int`, but returns a function `list -> list`)
- Imperative languages: pointers!
  - Structure with 2 fields (types "`int`" and "`list*`")
  - Second field: pointer to next element
  - Cannot be of type "`list`", $\rightarrow$ use "pointer to `list`"!

RDTs vs Pointers

- See? Imperative languages use pointers and explicit memory allocation...
  - Adding an element to list implies doing some malloc()/new for a node structure, setting some “next” pointers, etc...
- ...In functional languages, RDTs avoid the need for pointers, and memory allocation/deallocation is hidden...
  - Adding an element in front of a list “`l`” is as simple as “`let l1 = cons(e, l)`” or similar!
  - The implementation of the language abstract machine will take care of allocating memory, etc...