Generic Functions and Types or Parametric Polymorphism

Kenjiro Taura

2024/05/09

Motivation

say want to write …

- a function that ***sorts arrays of various types*** (e.g., ints, floats, strings, structs, …)
- a function that ***extracts elements from a list satisfying*** $p(x)$
- ***stacks, queues, trees, graphs, hashtables, etc.***
- many ***graph algorithms (breadth-first search, depth-first search, connected components, partitioning, etc.)***
- … ***without duplicating code*** for each element type

A trivial example (generic function)

write a generic function $f(a) = a[0]$ in your language (an element of an array) that works for any element type

Questions:

- do you have to specify the type of a?
- if so, how can you say ***a must be an array but whose element can be any type***
- if not, can it automatically apply to any array?
  - ***does it type-check statically*** (i.e., what if you pass something not an array)?

Type expressions

- things are conceptually straightforward
- but ***spelling out types*** needs a practice (for languages that require type annotations)
- master the syntax of ***type expressions, parameterized types/ functions, and instantiation thereof***

Type expressions for functions

ex. a type of ***functions taking an integer and returning a float***

Go	`func (int64) float64`	
Julia	`Function`	($*$), (†)
OCaml	`int -> float`	(†)
Rust	`fn (i64) -> f64`	

- ($*$) cannot specify input/output types
- (†) you normally don’t write it

Type expressions for array-like data

ex. (one-dimensional) array (or likes) of 64-bit floats

Go	fixed-size (n-element) array slice	`[n]float64` `[]float64`
Julia		`Vector{Float64}`
OCaml		`float array`

Rust	fixed-size (n-element) array
	vector
	slice
		`[f64; n]`
		`Vec<f64>`
		`[f64]`

Defining parameterized types

ex. `Node` (or `Tree`) of any type

Go	`type Node[T any] struct { ... }`
Julia	`struct Node{T} ... end`
OCaml	`type 'a tree = ...` `class ['a] node ... = object ... end`
Rust	`enum Tree<T> { ... }` `struct Node<T> { ... }`

Defining parameterized types

and a version parameterized by *any subtype of S*

Go	`type Node[T S] struct { ... }`
Julia	`struct Node{T<:S} ... end`
OCaml	not available
Rust	`enum Tree<T : S> { ... }` `struct Node<T : S> { ... }`

Instantiating parameterized types

ex. `Node` of 64-bit integers

Go	`Node[int64]`	
Julia	`Node{Int64}`	
OCaml	`int node`	
Rust	`Node<i64>` or `Node::<i64>`	(∗)

- (∗) `::` is necessary to disambiguate the symbol `<`
- ≈ `::` is unnecessary where only type expressions are expected and necessary when ordinary expressions are expected

Defining parameterized functions

ex. a function `dfs`, which can work for node of ***any type***

Go	`func dfs[T any](n Node[T]) { ... }`	
Julia	`function dfs(n : Node{T}) where T ... end`	
OCaml	`let dfs (n : 'a tree)  = ...` `let dfs n = ...`	(∗)
Rust	`fn dfs<T>(n : Tree<T>) { ... }`	

- (∗) : normally not necessary

Defining parameterized functions

and a version that can work for ***any subtype of S***

Go	`func dfs[T S](n Node[T]) { ... }`
Julia	`function dfs(n : Node{T}) where {T<:S} ... end`
OCaml	not available
Rust	`fn dfs<T : S>(n : Tree<T>) { ... }`

Instantiating parameterized functions

Go	bfs[int64](...)	
Julia	function bfs(...)	no specific syntax
OCaml	bfs ...	no specific syntax
Rust	bfs::<i64>(...)	