Programming Lanaugages (4) Parametric Polymorphism (aka Generic Types/Functions)

Kenjiro Taura

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.*
- variety of *graph algorithms (breadth-first search, depth-first search, connected components, partitioning, etc.)* that can/should work regardless of the exact data type of each node
- . . .

without duplicating code for each underlying type

A trivial example (generic function)

write a function

$$f(a) = a[0]$$

in your language (an element of an array, let's say)
Questions:

- do you have to specify the type of a?
- if so, how you can 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 pains are around *spelling out types*
- 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`
  - cannot specify input/output types
  - you normally don't write it
- OCaml : `int -> float`
  - you normally don't have to write it
- Rust : `fn (i64) -> f64`

Type expressions for array-like data

ex. (one-dimensional) array (or likes) of 64-bit floating point numbers

- Go :
  - n-element array: `[`n`]float64`
  - slice: `[]float64`
- Julia : `Vector{Float64}`
- OCaml : `float array`
- Rust :
  - n-element array : `[f64; `n`]`
  - vector : `Vec<f64>`
  - slice: `[f64]`

Defining parameterized types

ex. data type `node`, parameterized by *any type* `T` or `'a`

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

and a version parameterized by *any subtype of* S

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

Defining parameterized functions

ex. a function `bfs`, which can work for *any type*

- Go : `func bfs[T any](...) { ... }`
- Julia : `function bfs(...) where T ... end`
- OCaml : `let bfs ... =` (nothing special)
- Rust : `fn bfs<T>(...) { ... }`

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

- Go : `func bfs[T` S`](...) { ... }`
- Julia : `function bfs(...) where {T<:`S`} ... end`
- OCaml : not possible
- Rust : `fn bfs<T:`S`>(...) { ... }`

Instantiating parameterized functions

- Go : `func bfs[int64](...)}`
- Julia : `function bfs(...)`
- OCaml : `bfs ...` (nothing special)
- Rust : `fn bfs::<T>(...) { ... }`