Parametric Polymorphism
a.k.a. *Generic* Functions and Types

Kenjiro Taura

Motivations

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

Type-Parameterized Functions/Data

- what we need are functions or data structures that are *parameterized* by component type(s)
- something like:
  - $\alpha \rightarrow \text{int}$ (a function taking any type and returning int)
  - array of α (an array of any type)

What is an issue? — a trivial example

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, what does its type become?
  - take *anything*, array of *anything*, …?
  - ***does it type-check statically***?

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]`

- (∗) `n` has to be a compile-time constant

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>(...)`	

Type inference of compound expressions

- *type inference* generally refers to any algorithm that determines the static type of an expression without programmer's annotation
- virtually all languages infer types of compound expressions from their components
  - e.g., e_0 and e_1 are `int` $\Rightarrow$ $e_0 + e_1$ is `int`

Type inference of local variables

- many languages automatically infer types of local variables
  - Go : type of a local variable introduced by x `:=` e is inferred from e
  - Rust : type of a local variable introduced by `let` x `=` e is inferred from e
  - C++ has a similar mechanism (`auto`)
- *they all infer type of a compound expression from the types of its sub-expressions*
- for it to work, *type of function parameters must be given*

Type reconstruction (e.g., type inference in OCaml)

- OCaml's type inference is remarkable in that it infers types of function parameters from function body
- e.g.,
  - `let f x y = x + y` : $\text{int} \to \text{int} \to \text{int}$
  - `let f x = x` : $\alpha \to \alpha$
  - `let f a = a.[0]` : $\alpha\ \text{array} \to \alpha$
  - `let f a = a.[0] + 1` : $\text{int array} \to \text{int}$
  - `let f o = o#area < 1.0` : $< \text{area} : \text{float}; .. > \to \text{bool}$

How type reconstruction works

- intuitively, it works by collecting *constraints* and solving them
  - `x + y` → `x : int`, `y : int` (assumption: `+ : int -> int`)
  - `a.[i]` → `a : 'a array`
  - `s#area` → `s : < area : 'a; ... >`
- e.g., from `a.[i] + 1`
  - `a : 'a array`
  - `a.[i] : int` (because of `+`)
  - ⇒ `'a = int`

Remarks about OCaml

- thanks to type inference, you can omit most type annotations in OCaml, but it still does static type checking
  - not like Python or Julia
- it is indeed *type safe*
- to reason about what kind of programs are statically type-checked, you have to know possible types and their syntax (e.g., to understand error messages)
- it is possible, and indeed useful, to give type annotations for documentation and diagnosing type errors