# Dependent Types Paradigms

# Dependent Types Paradigms

**Unit-IV (15 Session)**

Session 6-10 cover the following topics:-

- *Dependent Type Programming Paradigm*- S6-SLO1
- *Logic Quantifier: for all, there exists*- S6-SLO2
- *Dependent functions, dependent pairs*– S7-SLO 1
- *Relation between data and its computation*– S7-SLO 2
- *Other Languages: Idris, Agda, Coq S8*-SLO 1
- Demo: Dependent Type Programming in Python S8-SLO2

*Lab 11: Dependent Programming ( Case Study) ( S8)*

**Assignment : Comparative study of Dependent programming in** *Idris, Agda, Coq*

**TextBook**:

1) Amit Saha, Doing Math with Python: Use Programming to Explore Algebra, Statistics,Calculus and More, Kindle Edition, 2015

**URL :**

- https://tech.peoplefund.co.kr/2018/11/28/programming-paradigm-and-python-eng.html
- https://freecontent.manning.com/a-first-example-of-dependent-data-types/

# Introduction

- A constant problem:
- Writing a correct computer program is hard
- Proving that a program is correct is even harder
- Dependent Types allow us to write programs and know they are correct before running them.

# What is correctness?

- What does it mean to be “correct”?
- Depends on the application domain, but could mean one or more of:
  - **Functionally correct** (e.g. arithmetic operations on a CPU)
  - **Resource safe** (e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . . )
  - **Secure** (e.g. not allowing access to another user’s data)

# What is Type?

- In **programming,** types are a means of classifying values
- Exp:  values 94, "thing", and [1,2,3,4,5]  □ classified as an integer, a string, and a list of integers
- For a ***machine***, types describe how bit patterns in memory are to be interpreted.
- For a ***compiler*** **or** ***interpreter***, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a ***programmer***, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

# Introductions

- In computer science and logic, a dependent type is a type whose definition depends on a value.
- It is an overlapping feature of type theory and type systems.
- Used to encode logic's quantifiers like "for all" and "there exists".
- Dependent types may help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.
- Exp: Agda, ATS, Coq, F*, Epigram, and Idris

# Dependent Type Example

- **Exp matrix arithmetic**
- **Matrix type** -□ refined it to include the number of rows and columns.
- Matrix 3 4 is the type of 3 × 4 matrices.
- In this type, 3 and 4 are ordinary values.
- A ***dependent type***, such as Matrix, is a type that's calculated from some other values.
- In other words, it *depends on* other values.
- **Definition**
  - A data type is a type which is computed from a ***dependent*** other value.

# Elements of dependent types

- **Dependent functions**
  - The return type of a dependent function may depend on the *value* (not just type) of one of its arguments
  - For instance, a function that takes a positive integer n may return an array of length n, where the array length is part of the type of the array.
  - (Note that this is different from polymorphism and generic programming, both of which include the type as an argument.)
- **Dependent pairs**
  - A dependent pair may have a second value of which the type depends on the first value

# Formal definition

## $\Pi$ type [edit]

Loosely speaking, dependent types are similar to the type of an indexed family of sets. More formally, given a type $A : \mathcal{U}$ in a universe of types $\mathcal{U}$, one may have a **family of types** $B : A \to \mathcal{U}$, which assigns to each term $a : A$ a type $B(a) : \mathcal{U}$. We say that the type *B(a)* varies with *a*.

A function whose type of return value varies with its argument (i.e. there is no fixed codomain) is a **dependent function** and the type of this function is called **dependent product type**, **pi-type** or **dependent function type**.[3] For this example, the dependent function type is typically written as

$$\prod_{x:A} B(x)$$

or

$$\textstyle\prod_{x:A} B(x).$$

If $B : A \to \mathcal{U}$ is a constant function, the corresponding dependent product type is equivalent to an ordinary function type. That is, $\prod_{x:A} B$ is judgmentally equal to $A \to B$ when *B* does not depend on *x*.

# Formal definition

## $\Sigma$ type [ edit ]

The dual of the dependent product type is the **dependent pair type**, **dependent sum type**, **sigma-type**, or (confusingly) **dependent product type**.[3] Sigma-types can also be understood as existential quantifiers. Continuing the above example, if, in the universe of types $\mathcal{U}$, there is a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$, then there is a dependent pair type

$$\sum_{x:A} B(x).$$

The dependent pair type captures the idea of an ordered pair where the type of the second term is dependent on the value of the first. If

$$(a, b) : \sum_{x:A} B(x),$$

then $a : A$ and $b : B(a)$. If $B$ is a constant function, then the dependent pair type becomes (is judgementally equal to) the product type, that is, an ordinary Cartesian product $A \times B$.

# Pseudo-code

- **General Code**

```
float myDivide(float a, float b)
 { if (b == 0)
return ???;
Else
 return a / b;
 }
```

- **Dependent Type Code**

```
float myDivide3
(float a, float b, proof(b != 0)
    p)
{
return a / b;
 }
```

**Auto Checking done here**

# Python Simple Example

from typing import Union def return_int_or_str(flag: bool) -> Union[str, int]:

if flag:

return 'I am a string!‘

return 0

# Dependent Type

- » pip install mypy typing_extensions
- from typing import overload
- from typing_extension import Literal

**Literal**

Literal type represents a specific value of the specific type.

```
from typing_extensions import Literal
 def function(x: Literal[1]) -> Literal[1]:
 return x
function(1)
# => OK!
function(2)
 # => Argument has incompatible type "Literal[2]"; expected "Literal[1]"
```

# Python Example

x :: Iterator[T1]

y :: Iterator[T2]

z :: Iterator[T3]

product(x, y) :: Iterator[Tuple[T1, T2]]

product(x, y, z) :: Iterator[Tuple[T1, T2, T3]]

product(x, x, x, x) :: Iterator[Tuple[T1, T1, T1, T1]]

- All the above replaced by
  - def product(*args :Tuple[n]) -> Iterator[Tuple[n]]: pass

# A first example: classifying vehicles by power source IDRIS Exampl

**Listing 1** Defining a dependent type for vehicles, with their power source in the type (vehicle.idr)

```
data PowerSource = Petrol | Pedal                ❶
data Vehicle : PowerSource -> Type where             ❷
Bicycle : Vehicle Pedal                    ❸
Car : (fuel : Nat) -> Vehicle Petrol          ❹
  Bus : (fuel : Nat) -> Vehicle Petrol          ❹
```

❶ An enumeration type describing possible power sources for a vehicle
❷ A Vehicle’s type is annotated with its power source
❸ A vehicle powered by pedal
❹ A vehicle powered by petrol, with a field for current fuel stocks

# IDRIS Second Example

**Listing 2** Reading and updating properties of Vehicle

wheels : Vehicle power -> Nat ❶

wheels Bicycle = 2 wheels (Car fuel) = 4 wheels (Bus fuel) = 4

refuel : Vehicle Petrol -> Vehicle Petrol ❷
refuel (Car fuel) = Car 100 refuel (Bus fuel) = Bus 200

❶Use a type variable, power, because this function works for all possible vehicle types.
❷ Refueling only makes sense for vehicles that carry fuel. Restrict the input and output type to Vehicle Petrol.

# References


- http://www.cs.ru.nl/dtp11/slides/brady.pdf
- https://freecontent.manning.com/a-first-example-of-dependent-data-types/
- https://en.wikipedia.org/wiki/Dependent_type
- https://livebook.manning.com/book/type-driven-development-with-idris/chapter-1/13
- https://github.com/python/mypy/issues/366