Unit IV
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.
- **Dependent Type:** It is a concept when you rely on values of some types, not just raw types.

Dependent Type Programming Paradigm

- 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.
- Example languages: Agda, ATS, Coq, F*, Epigram, and Idris

Dependent Type Example

- **Consider 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.
  - 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
  - With the array example, a dependent pair may be used to pair an array with its length in a type-safe way.

Pseudo-code

- **General Code**

```
float myDivide(float a, float
    b)
 { if (b == 0)
return 0;
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
```

Literal and Overload

- » 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]"
```

Literal

- Difference between Literal[0] and int type

```
from typing_extensions import Literal
def function(x: int = 0, y: Literal[0] = 0) -> int:
    reveal_type(x) # => Revealed type is 'builtins.int'
    reveal_type(y) # => Revealed type is 'Literal[0]'
    return x
```

- Revealed types differ. The only way to get Literal type is to annotate is as Literal.
- You can use Literal[0] everywhere where a regular int can be used, but not the other way around.

Overload Decorator

- It is required to define multiple function declarations with different input types and results.
- Example: to write a function that decreases a value.
  - It should work with both str and int inputs.
  - When given str it should return all the input characters except the last one, but when given int it should return the previous number.

Example for @overload

```
from typing import Union, overload
@overload
def decrease(first: str) -> str:
    """Decreases a string."""
@overload
def decrease(first: int) -> int:
    """Decreases a number."""
def decrease(first: Union[str, int]) -> Union[str, int]:
    if isinstance(first, int):
        return first - 1
    return first[:-1]
reveal_type(decrease(1))          # => Revealed type is 'builtins.int‘
reveal_type(decrease('abc')) # => Revealed type is 'builtins.str'
```

Example

- Consider open function from the standard library

```
def open_file(filename: str, mode: str):
    return open(filename, mode)
```

- Here the return type is Union[IO[str], IO[bytes]].
- Dependent types solve this problem.
- Solution:
  - we need to return bytes for 'rb' mode and str for 'r' mode.
  - we need to know the exact return type.

Example

- Algorithm :
  - Write several @overload decorators to match all possible cases
  - Write Literal[] types when we expect to get 'r' or 'rb'
  - Write function logic in a general case

```python
from typing import IO, Any, Union, overload
from typing_extensions import Literal

@overload
def open_file(filename: str, mode: Literal['r']) -> IO[str]:
    """When 'r' is supplied we return 'str'."""

@overload
def open_file(filename: str, mode: Literal['rb']) -> IO[bytes]:
    """When 'rb' is supplied we return 'bytes' instead of a 'str'."""

@overload
def open_file(filename: str, mode: str) -> IO[Any]:
    """Any other options might return Any-thing!."""

def open_file(filename: str, mode: str) -> IO[Any]:
    return open(filename, mode)

reveal_type(open_file('some.txt', 'r'))
# => Revealed type is 'typing.IO[builtins.str]'
reveal_type(open_file('some.txt', 'rb'))
# => Revealed type is 'typing.IO[builtins.bytes]'
reveal_type(open_file('some.txt', 'other'))
# => Revealed type is 'typing.IO[AnyStr]'
```

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                    (1)
data Vehicle : PowerSource -> Type where             (2)
Bicycle : Vehicle Pedal                          (3)
Car : (fuel : Nat) -> Vehicle Petrol             (4)
  Bus : (fuel : Nat) -> Vehicle Petrol           (4)
```

(1) An enumeration type describing possible power sources for a vehicle
(2) A Vehicle's type is annotated with its power source
(3) A vehicle powered by petrol
(4) 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
```

❶ Use 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
- https://www.idris-lang.org
- https://pypi.org/project/dependent-types/
- https://dev.to/wemake-services/typeclasses-in-python-3ma6