Using types of rule out bugs: *Python* vs *Fortran* perspective

Dominic Orchard

WCRP OSC - ICCS training day - 21st October 2023, Kigali, Rwanda

Warmup!

```
1 + 1 = 2
```

```
"hello" + 1  = "hello1"
             = “iello"
             = "hellp"
```

JS

asm

??!

```
"hello" * 2  = "hellohello"
```

python™

```
"hello" / 2  = ???
```

🤔

Fortran rejects all but the first

because the *types* of the inputs don’t match what +, *, / expect

Types communicate to us what the computer can do

Learning objectives

- Understand key ideas behind specification and verification
- Understand some key concepts and terminology behind types
- Compare and contrast power of types in Fortran and Python

  Why these two? Used a lot in climate science. But ideas transferable.

- Learn about the mypy tool for typing in Python
- Develop ability to use types to avoid bugs and write code more effectively

Validation

Did we implement the right equations?

vs

Verification

Did we implement the equations right?

Challenge

Telling these two apart when results are not as expected

A helpful model: types as sets

- Set defined by its elements (*data), e.g.,*
  - $\mathbb{N}$ - Natural numbers $\{1, 2, \ldots\}$ or $\{0, 1, 2, \ldots\}$ depending who you ask!
  - $\mathbb{Z}$ - Integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - $\mathbb{R}$ - Real numbers $\{\ldots, 0, 0.1, 0.11, \ldots, e, \ldots, \pi, \ldots\}$
- Sets of pairs of A and B *written* $A \times B$ (Cartesian product)
  - e.g., $\mathbb{N} \times \mathbb{N} = \{(1,1), (1,2), (2,1), (2,2), \ldots\}$
- Functions from A to B *written* $A \to B$
  - e.g. $\mathsf{abs} : \mathbb{Z} \to \mathbb{N}_0$
  - $\sqrt{\ } : \mathbb{R}_{\geq 0} \to \mathbb{R} \times \mathbb{R}$
  - $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

Notational convention

expression : type

type signature / specification

Static typing vs. Dynamic typing

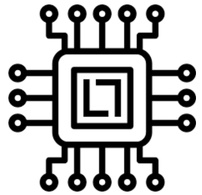
- Compiler first does type checking
- Ill-typed programs rejected
  - Intrinsic typing - *Ill-typed programs have no meaning* (cannot be run)
- Well-typed programs compiled, using types for optimisation

- No pre-run checks
- Data stored with type information
- Operations check type information
- Errors occur “as it happens”

- Today: we will use mypy to add static typing to Python

Without types?

- E.g., in *assembly languages*



  - *One type* = bits!
  - Everything works / operations may not do what you want
  - *Developer has to track meaning themselves*

Types eliminate a class of bugs

“Well typed programs cannot go wrong”
(Milner, 1978)

(For some definition of wrong!)

Fortran primitive types

	Possible kinds (default highlighted)
`integer`	`1,2,4,8,16`
`real`	`4,8,10,16`
`logical`	`1,2,4,8,16`
`character`	`1,4`
`complex`	`4,8,10,16`

`4`=*float (32-bit IEEE-754),* `8`=*double*

- Each can have ‘kind’ parameter to specify number of bytes

  e.g., `real(kind=8)`

- All can be used as the element type of an array by adding `dimension` modifier

  e.g., `real,dimension(1:10)` (*type of a 10-element 1D floating-point array)*

  e.g., `integer,dimension(20,30)` (*type of a 20×30-element 2D integer array)*

Fortran derived data types

(Like `struct` in C, dictionary in Python, or records in other languages)

- Define a new type name
- Comprising types combined as big product of types (i.e. $A \times B \times \ldots$)
- For example:

```
type Coords
  real :: lat
  real :: lon
  real :: sigma
end type Coords
```

← Derived DT definition

```
type(Coords) :: origin
```

← Declare type signature

```
origin%lat = 0.0
origin%lon = 0.0
origin%sigma = 0.0
```

← Access "fields" `var%name`

Type casts

Convert from one type to another

- Usually provided by a conversion function if the data format changes:

  - e.g., INT intrinsic function converts to integer, REAL converts to real
  - See https://fortran-lang.org/en/learn/intrinsics/type/

- Unsafe type casts do no conversion

  - TRANSFER intrinsic just copies the bits with no conversion
  - *(code demo)*

mypy

An optional gradual, static type system for Python

- Gradually convert from dynamic to static typing
- Optional $\Longrightarrow$ *extrinsic typing* - ill-typed programs can still run (have meaning)
- Maths-like *type signatures*

```
flag : bool = True
```

```
def plus(x : int, y : int) -> int:
    return x + y
```

Getting mypy (if you want to ‘code along’)

```
python3 -m pip install mypy
```

Or possibly:

```
python -m pip install mypy
```

Mypy/Python primitive types

`int`	
`bool`	
`float`	
`str`	
`None`	*(no result type)*
`Any`	*(fall-back, anything)*

```python
def greet(name: str) -> None:
    print("Hi " + name)
```

(compare with Fortran subroutines)

Type constructors

Like *type functions*: create a type from other types

- For some type `t` then `list[t]` captures lists of elements (all) of type `t`

```
def greet_all(names: list[str]) -> None:
    for name in names:
        print('Hello ' + name)
```

- `tuple[t1, t2, …]` captures tuples with elements of type t1, t2, etc.

```
some_data : tuple[int, bool, str] = (42, True, "Kigali")
```

Type constructors

Like *type functions*: create a type from other types

- `dict[k, v]` captures records/dictionaries of key k and value v type:

```
x: dict[str, float] = {"field1": 2.0, "field2": 3.0}
```

- `t1 | t2` captures either type t1 or t2 type (Python 3.10 <= `Union[t1, t2]`)

```
def myDiv(x : float, y : float) -> (float | None):
    if y != 0: return x / y
    else:      return None
```

Querying mypy

Ask mypy what it *infers* the type to be:

```
reveal_type(expression)
```

Subtyping

- In theory literature, A is a subtype of B written $A :< B$ (*think subsets*)
- Example: `list[t]` is a "subtype" of `Iterable[t]`
- Can pass arguments of a subtype to a function

$$\frac{x : A \qquad f : B \to C \qquad A :< B}{f(x) : C}$$

e.g.

```
def greet_all(names: Iterable[str]) -> None:
    for name in names:
        print('Hello ' + name)

names = ["Alice", "Brijesh", "Chenxi"]
greet_all(names)   # Ok!
```

Polymorphism

(Also known as *generic types*)

- Consider the function

```
def first(xs : list[str]) -> str:
    return xs[0]
```

- What if we want to use it with list[int] too?

```
def first_int(XS : list[int) -> int:
    return xs[0]
```

- **Duplication bad for maintenance and understanding**

Polymorphism

(Also known as *generic types*)

- Solution: generalise to any element type T

```
T = TypeVar('T')

def first(xs : list[type[T]]) -> type[T]:
    return xs[0]
```

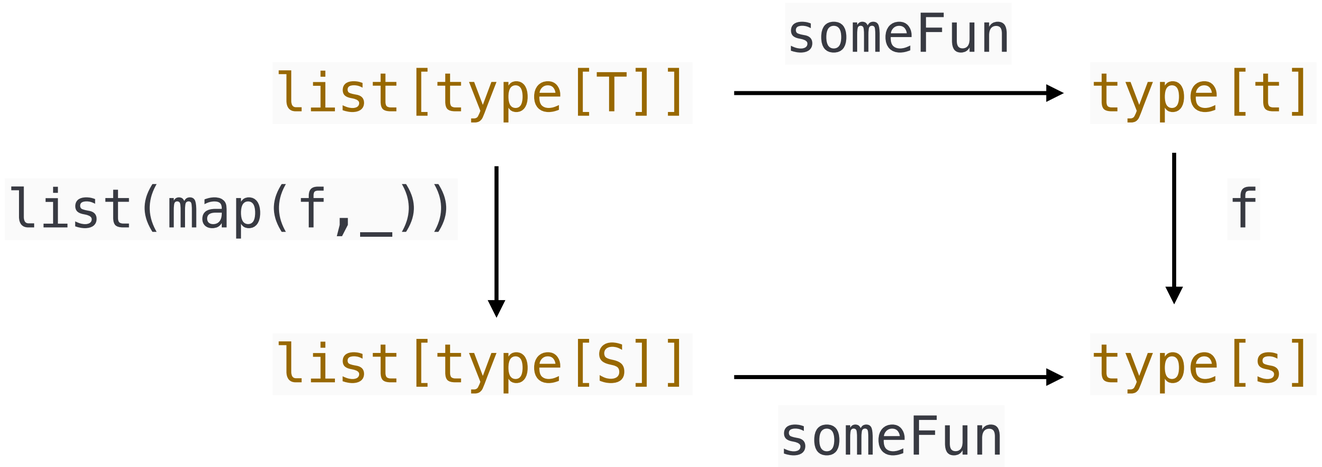
- (Note: requires an import)

```
from typing import TypeVar, Generic
```

“Free theorems” follow from polymorphic types

- Consider `def someFun(XS : list[type[T]]) -> type[T]`
- “Universality” of `T` tells us we cannot inspect or compute with the `T` elements
- Implies the following (“*naturality*”) property:

```
someFun(list(map(f, x))) = f(someFun(x))
```



Note the right expression applies `f` once, the left applies it `len(x)` times.

Optimisation!

Things we don’t have time to cover: Overloading

(aka *ad hoc polymorphism*, or in OOP *polymorphism*)

- Functions that work on different types but different behaviour per type
- Also for functions with different arity

naryfunc.f90

```
program example
  use naryfunc
  implicit none
  ! Outputs 2 6 24
  write(*,*) mult(2), mult(2, 3), mult(2, 3, 4)
end program example
```

Coming into land….
What did we learn?

- Understand key ideas behind specification and verification
- Understand some key concepts and terminology behind types
  - “Sets” model
  - Static vs dynamic
  - Extrinsic vs intrinsic
  - Polymorphism
  - Subtyping

Coming into land….
What did we learn?

- Learn about the mypy tool for typing in Python
- Compare and contrast power of types in Fortran and Python
  - mypy gives us extrinsic static typing
  - Fortran $<$ 2003 has no polymorphism (Polymorphism via OOP in Fortran $\geq$ 2003)
- Develop ability to use types to avoid bugs and write code more effectively
  - Go and practice on your own and start using in projects

Thanks- and happy typing!

https://dorchard.github.io

types.pl/@dorchard

@dorchard