**Formal definitions:**

Type is “In [computer science](https://en.wikipedia.org/wiki/Computer_science) and [computer programming](https://en.wikipedia.org/wiki/Computer_programming), a **data type** (or simply **type**) is a set of possible values and a set of allowed operations on it” (<https://en.wikipedia.org/wiki/Data_type#Definition> )

Type is {DataValues, OperationOnDataValues}

Example:

{{1,2,3}, {+,-,\*,/}} data values are 3 integer numbers and set of operations with names explicitly stated

Integer is {{MinInteger..MaxInteger}, {{+,-,\*,/}}} here we name the type and define its values as a range plus names of operations

In both cases we still have a lot of items which are not fully defined. We have not defined fully operations – we just named them. And we have not defined the menaing of MinInteger and MaxInteger as well as what 1, 2, and 3 are. Also we have not defined what range is.

**type** T **is** {{**new** T(arguments1), **new** T(arguments2)}, {T.routine1, T.routine2}}

One more attempt to declare type T – let’s try to put all possible variants of how we crate instances (objects) of type T and provide a full list of routines (procedures and functions) which compromise set of operations over objects of type T. Of course we may drop T and new inside curly brackets

**type** T **is** {{(arguments1), (arguments2)}, {routine1<signature1>, routine2<signature2>}}

So, what we have got – list of all possible calls to constructors (initialization procedures) of type T plus all routines with their signatures. What we cannot list fully is all combinations of constructor calls. So, we need another mechanism of specifying which objects will be valid for the type T taking into account that we do not like to list them all as a general mechanism. And we have to look deeply at what object (instance of some type) is. If we take one element of the possible type data values and add set of allowed operation we have the object. So, set of all objects is the type of these objects. They all have the same operations while different values.

{1, {{+,-,\*,/}}} - that is object 1

{2, {{+,-,\*,/}}} - that is object 2

Can we use the value of the object as it unique identifier – yes we can. So, 1 is both the name and value of the object. If we have a complex number then the scheme stays the same. 5+6i can be written as 5i6 or (5,6) but 5 as real part and 6 as imaginary one form the value and name of such object.

So, intuitively we see that every object has some internal structure – list of fields or attributes which contains next level values. And these values have types as well. So, if to drill down and reach the end of this hierarchy we have only 2 atomic objects bar and circle, | and O, or 1 and 0. All other objects are constructed from these constituents. What is the type objects 1 and 0 belong to? It was named as Bit. So, here is the first essential definition

**type** Bit **is** {{0, 1}, {set of operations over bits}}

Can we express it in a form of constructors?

**type** Bit **is** {{zero(), one()}, {set of operations over bits}}

Why not to name constructor? And we did it.

But let’s make a step back to fields or attributes of an object. The most straightforward way is just to give a name to each attribute

MyObject1 **is** {{f1 **is** value11, f2 **is** value12}, {set\_of\_operations}}

MyObject2 **is** {{f1 **is** value21, f2 **is** value22}, {set\_of\_operations}}

…

MyObjectn **is** {{f1 **is** valuen1, f2 **is** valuen2}, {set\_of\_operations}}

So, type of all these objects is a matrix. That is another representation of type – matrix of all type instances. But if to look more deeply at operations we may notice that operation are just constant attribute of the routine type. Let’s see a brief example

foo (p1: T1; p2: T2):RT …

**const** foo **is routine** (p1: T1; p2: T2):RT …

So, notation which is widely used to denote functions is a kind of shortcut for the constant attribute when name of the function will be the name of the this attribute. And then every object is the vector of attributes and type is the matrix of them. All vectors are of the same length

AnyObject1 **is** {f1 **is** value11, f2 **is** value12 …, fn **is** value1m}

AnyObject2 **is** {f1 **is** value21, f2 **is** value22 …, fn **is** value2m}

…

AnyObjectn **is** {f1 **is** valuen1, f2 **is** valuen2 …, fn **is** valuenm}

Next logical step is to state that every column (value1i, value2i, value3i, …, valueni) defines some typei in fact. So, from matrix of objects we come to the vector with types

**type** TypeOfAnyObjects **is** {f1 **:** Type1, f2 : Type2 …, fn : Typem}

That fully explains the notation which is widely used for structures, records, classes, interfaces and so on. Some particular type kinds will have empty set of values are called interfaces, some will have operations with no real, effective body and they are called abstract classes. Both may not have objects created. But the key thing is that all such types have a complete description of type attributes and routines. Does this kind of type deserve a special name? Well the name which suits is class. The tricky thing is that for years the term class was used together with association that all data entities of this type either reference kind variables (Java and many other languages) or value ones (C++). To break this contradicting association the term unit is proposed to denote a type which has an explicit description of its attributes, operations, and construction procedures. So, unit type itself has no direct correlation with the form of data entities

As for the set of values for the objects of MyType – how it can be defined. As product of value sets of type of every attribute of MyType. values(MyType) is P (values(Typei)) for i in 1..m

Are there type kinds which do not have an explicit description of its attributes and operations (routines)? Yes they are. Algebraic types are the first example.

**type** Union **is** T1 + T2 + T3 // union of types

**type** Intersection **is** T1 \* T2 \* T3 // intersection of types

One more group of types is tuple types

**type** Tuple **is** (T1, T2, T3)

Another one is anchored ones

**type** Anchor1 **is like this**

**type** Anchor2 **is like** someAttribute

Function (routine) types will look like

**type** Function **is** **routine** (Par1Type, Par2Type): ReturnType

**type** Procedure **is** **routine** (ParType)

They also define a set of values and set of operations but they do not list them explicitly. And another distinction between union (class) types and all others those union types explicitly state their inheritance relation (extend-implements one in Java terms) to other union types

It leads us to an alternative definition of the term type. Type is a set of objects which belong to this type.

Integer is -2\*\*31+1..2\*\*31-1

Color is Red, Green, Blue

Integer | Color is -2\*\*31+1..2\*\*31-1 | Red | Green | Blue

So, it is worth to define what inheritance is. Short and 100% correct definition it is relation between union types. But this does not reflect the nature of this relation. And the nature can be expressed that if B inherits A then B will have some attributes and routines from B, some adopted and some new introduced immediately in B. So, for convenience we will be using the C++ terminology calling unit attributes and routines as members. Member attribute and member routine

Well is it possible that derived (child) unit will have just a subset of base (parent) unit members. Why not? What we cannot do in this case we cannot use object of derived type instead of the object of the base one. Effectively we cannot assign to the object of the derived unit to the base one. So, in other word derived unit does not conform to the base one. It just have a different form or objects. This brings us to the concept of conformant inheritance and non-conformant one. But in any case inheritance means a kind of movement of members of base classes to the derived one

Another important concept which was already mentioned is reference and value entities. When we say object we always mean set of values and set of operations-routines. So, we may notice that we treat object as a value data entity. So, what is a data entity or simply an entity? Local (declared within some routine) variable or constant, unit variable or constant attribute, this, function result are examples of entities. All these entities can be in the form of reference to an object or an object itself (value). And they all have names associated with these entities. Objects can have no name or treat set of its values as its name! Entity is part of notation used by programmer which creating the source code. Objects exit while program is being executed and can be stored not only in RAM but on external storage too. Entities are abstractions expressed in the source code.

When we deal with entities we have only 2 operations – entity creation and ‘dot-call’ operation

a **is** A or a **is** **new** A or a **is** **new** A() syntax form is just a sugar

a := expr0

a.foo (expr1, expr2)

b := struct.field

struct.field := expr3

Can we interpret assignment as a form of the ‘dot-call’ – yes!

a.:= (expr0)

b.:=(struct.field)

struct.field.:=(expr3)

So, entity has 2 states – created and not-created-yet. There are cases when we need to declare an entity but not to create it immediately. Keep the name but to have it not attached to a real object. Be non-initialized. (That is null-safety one may exclaim – let it be so, but conceptually different)

So, based on the declaration of an entity in the program source code entities could be attached to an object and detachable (may be either attached to an object or not). How object can be attached to an entity. It can be attached by the runtime of the program like ‘this’ is automatically attached to the currently active object. Or entity declaration may have explicit (a **is new** A) or implicit (a: A) initialization which attaches this entity to some object. Or it can be a direct assignment of freshly created object or already initialized entity to the entity of interest (a := **new** A; a := b)

As it was already shown that type is fully defined when we define all its values and all its operations. Let’s explore ways how we can defines all values – all possible instances of the type

1. We may explicitly name them all
2. We may provide a kind of regular expression or generating function which will generate all objects
3. We may provide a set of predicates which describe which objects are valid ones for the particular type

Let’s see how it works

**type** Color **is** (Red, Green, Blue) /\* Red, Blue and Green are objects (better say constant objects) of type Color \*/

**type** SomeNumbers **is** (1, 15, 463)/\* Type SomeNumbers is the type which has only 3 integer data values \*/

**type** SomeRange **is** 1..123 /\* Values of type SomeRange are defined as a generator which starts from integer value 1 and next value is built adding 1 to the previous one until we reach 123 inclusively \*/

**type** SomeOdds **is** 3{+2}..33 /\*Odd numbers starting from 3, next one will be generated adding 2 to the previous one until we reach 33\*/

**type** SomeType **is** Integer **ensure** **this** < 1234 /\*Here we put one or more predicates which state some kind of Boolean expression which allows to filter which object will be a valid value for the type SomeType\*/

And now let’s try to see how to introduce such filters for the unit types. Unit is a set of attributes as we defined above.

What is an attribute?

Data entity – logically and physically separate element within the object or routine. Object attributes are in fact variable or constant fields of this object. While local variables of constants of any routine are local attributes. Routine parameters are local attributes of the routine as well. So, as any routine is a function (routine) object then it local attribites are attributes of the routine object.

What are the unique characteristic of an attribute?

* Name
* Type (empty for procedures)
* Signature where signature for
  + a constant is the value of this constant
  + a variable is empty
  + a routine is a tuple of types of its parameters

So, two attributes are identical if they have identical name, type and signature. If at least name or type or signature is different then these two attributes are different. We can distinguish them and compiler can do the same

atrr1: Type1 // That is the variable

attr2(): Type2 // That is a function

attr3(): Type3 // That is another function

**const** attr1: Type1 /\* That is duplicating definition as it has the same name, type and signature (empty) as attr1 declared first \*/

Attributes may be of two kinds – reference or value kind. A reference attribute store a reference to an object it is attached to. A value attribute has no extra reference and it is an object itself if it is attached to an object and just a name if it is not attached (but memory space may be reserved)

attr1: **ref** Type /\* attr1 is the reference kind attribute of type Type always attached to an object\*/

attr2: **val** Type /\* attr2 is the value kind attribute of type Type always attached to an object\*/

attr3: **ref** **?**Type /\* attr3 is the reference kind attribute of type Type which may be detached \*/

attr4: **val ?**Type /\* attr4 is the value kind attribute of type Type which may be detached \*/

So, attr1 and attr2 can be valid declarations only if the attribute gets its initial value – attached to a valid object before any activation can be applied to this attribute. Such initialization can be done by one of three ways

1. Type has initialization procedure with no parameters – then i: Integer will lead to the case that I will be attached to an object of Integer type with the value 0, due to the fact that Integer is declared like this

**val unit** Integer {

Integer () { …. } // ‘Default’ Integer constructor

{

1. Inline explicit initialization

attribute: Type **is** expression /\* type of expression should be compatible with the Type \*/

or with type inference

attribute **is** expression // type of expression will be the type of the attribute

1. Initialization of the object attribute is done within the body of the object type initialization procedure (constructor)

**unit** SomeUnit {

attribute: Type

SomeUnit (…) {

…

attribute **is** expression

…

}

}

Consequence of signature definition – overloading

Overloading will work for all attribute kinds – for routines and data entities

procName (parameters1) {…}

procName (parameters2) {…}

funcName (parameters1): T1 {…}

funcName (parameters2): T2 {…}

varName: T1

varName: T2

All examples above are valid examples of attribute name overloading. If the actual version of an attribute can be uniquely derived from the context then compiler will do it if several versions are available then programmer has explicitly specify which one is to be activated. See below

procName (arguments)

if types of arguments match only parameters1 then it means that it should be called. If arguments types match both parameters1 and parameters2 then call will be valid if explicit types are provided to ensure only version matches

procName (e1, e2, … en) -> (E1, E2, … , En) E1, … Ek is compatible with T1, … Tk and with U1, … Uk

procName (p1: T1, p2: T2, … pn: Tn)

procName (p1: U1, p2: U2, … pn: Un)

Then for arguments 1..k call should be written as

procName (e1, …, [Tj|Uj] ej, … [Tj+k|Uj+k] ej+k, … en)

Particular example will look like

**unit** Base {}

**unit** D1 **extend** Base {}

**unit** D2 **extend** Base {}

**unit** D3 **extend** D1, D2 {}

procName (p1: D1)

procName (p1: D2)

procName (**new** D3) // Here we have two versions available for the call

procName ([D1]**new** D3) // Explicitly tell to compiler which one to call

procName ([D2]**new** D3) // Explicitly tell to compiler which one to call

in case of functions and data entities the same scheme is to be used adding typer in front of the activation

**unit** A {

attr: D1

attr: D2

}

localAttr **is** A.attr // Ambiguity as we have 2 versions

localAttr **is** [D1] A.attr // Ambiguity can be resolved attaching the type to the expression

localAttr: D1 **is** A.attr // or to the context

procName (A.attr) // to be replaced with

procName ([D1] A.attr)

procName ([D2] A.attr)

xxxx

Incidence matrix and attribute decorated inheritance graph

Matrix is one of form how graph can be represented. When we consider inheritance we typically focus on relation between classes only and then incidence matrix will reflect existence or absence of the edge between two classes. See example below

**unit** Base **end**

**class** Derived **extend** Base, Base **end**

|  |  |  |
| --- | --- | --- |
|  | Base | Derived |
| Base |  |  |
| Derived | + |  |

But if we make next step and put into consideration unit attributes then we will have a bit different picture

**unit** Base

foo

**end**

**class** Derived **extend** Base, Base

**end**

|  |  |
| --- | --- |
|  | foo$Base |
| Base | foo@Base |
| Derived | foo@Base |

Every column in the table reflects all attributes which were declared in all units and all rows are units which we have in our system. Notation attributeName$unitName stands for seed and origin, where seed is the name of the attribute and origin is the name of the unit where such attribute was declared. Note that overriding an attribute does not create new declaration – it is re-declaration in fact. Notation attributeName@unitName refers to a particular implementation of an attribute which was done in some unit – so first declaration and every re-declaration will issue new pair attributeName@unitName

Technically we may consider attributeName@unitName as routine address if the attribute is unit routine and offset from this if attribute is unit variable

**abstract unit** A

foo

g00 abstract

**end**

**unit** B **extend** A

**override** foo

**override** goo

**end**

|  |  |  |
| --- | --- | --- |
|  | foo$A | Goo$A |
| A | foo@A |  |
| B | foo@B | goo@B |