Unified type system (UTS) for the modern general-purpose programing language

Alexey Kanatov

Eugene Zouev

Innopolis University, Innopolis, Tatarstan, Russia

Agenda for 20 min

- Personal introduction (5 sec ☺) – use Facebook or LinkedIn
- UTS Introduction - from dust to heaven (from atoms to molecules) (3 min)
- Type kinds (9 min)
- Compatibility = conformance + convertibility (2 min)
- Type test
  - Duck typing (1 min)
- Summary (5 sec ☺)

You could have 4 min 50 sec for Q&A ☺ at the end please!

UTS Introduction: from atoms to molecules

- **I think OOP**: there is nothing except concurrent interacting objects in the world☺ Others think only functions exist ☺
- **Object**: region(s) in the computer memory where attributes and routines are stored. At compile time it has the name (entity placeholder), offset and size at runtime
- **Objects' hierarchy**: Real world object => abstraction => computer object => description of the object (another abstraction) => type (values+operations). Implication => there are no types at runtime (only type descriptions as objects)
- **2 fundamental objects** – O and I, 0 and 1, 0b and 1b, 0b0 and 0b1. All other objects are built from these 2.
- What is their type => **Bit**. Type Bit has 2 constant objects Bit.0b0 and Bit.0b1; For bits name and value are the same

UTS Introduction: from atoms to molecules

```
val unit Bit const: 0b0, 0b1 end
  pure & alias and (other: Bit): Bit => if this = 0b0 do 0b0
elsif other = 0b0 do 0b0 else 0b1
  pure| alias or (other: Bit): Bit => if this = 0b1 do 0b1 elsif
other = 0b1 do 0b1 else 0b0
  pure ^ alias xor (other: Bit): Bit => if this = other do 0b0
else 0b1
  pure ~ alias not (): Bit => if this = 0b0 do 0b1 else 0b0
  pure + (other: Bit): Bit do
    if this = 0b0 do return other
    elsif other = 0b1 do raise "Bit overflow"
    else return 0b1 end // if
  end // +
  pure - (other: Bit): Bit do
    if this = other do return 0b0
    elsif this = 0b1 do return 0b1
    else raise "Bit underflow“end // if
 end // -
end // Bit
```

entity

object

```
b1: Bit is Bit.0b0
b2 is Bit.0b1
b1 := b1.and (b2)
b1 := b1 and b2
b1 := b1 & b2
b1 := b1.&(b2)
```

UTS Introduction: from atoms to molecules

```
val unit Integer extend Integer [Platform.IntegerBitsCount]
  …
end
val unit Integer [BitsNumber: Integer] extend Numeric,
Enumeration
   const minInteger is - (2 ^ (BitsNumber - 1))
   const maxInteger is 2 ^ (BitsNumber - 1) - 1
   const: /* That is ordered set defined as range of all Integer
constant values (objects) */
      minInteger .. maxInteger
   end
   init do
      data is Bit [BitsNumber]
   end
   {this} data: Bit [BitsNumber] // private
require
   BitsNumber > 0 /* Number of bits in Integer must be greater
than zero! */
end
```

Type kinds (7+1)

1. **Unit-based:** the type which has a full textual description of all its members => `unit A … end`
2. **Anchored:** the type which is the same `as` the other entity type
3. **Multi-type (ADT +):** entity of this type may be of type T_1 or T_2 or … => `T1 | T2 | … Tn`
4. **Tuple type (ADT *):** group => `(T1, T2, … Tn)`
5. **Range type:** explicitly name values => `1..6` or `1 | 17 | 2..3` or `a .. b` or `a | b .. c`
6. **Routine(function) type**: signature is essential here => `rtn (T1, T2): T3`
7. **Unit**: type as 1st class citizen => `Type: unit … end` `attr: Type`
8. ***Detachable type*:** the entity has no value but its static type is known at compiler time => `?Integer` or `?AnyType`

Type kinds: unit-based

```
unit UnitNameIsTheTypeName
      const constant1: Type is someExpression
      const constant2 is someExpression
      attribute: Type
      methodProcedure do
            methodConstant1: Type is someExpression
            methodConstant2 is someExpression
            var methodVariable1: Type is someExpression
            var methodVariable2 is someExpression
      end
      init do
            attribute is someExpression
      end
end
virtual unit Any use const Integer, Real, Boolean, Character,
Bit, String
  …
end // Any
```

Type kinds: anchored

```
virtual unit Any …
   /// Shallow equality test
   = (that: as this): Boolean foreign
   final /= (that: as this): Boolean => not ( this = that)
   …
end // Any

unit System
   clone (object: Any): as object foreign ///* Shallow version of the object clone operation */
   deepClone (object: Any): as object foreign ///* Deep version of the object clone operation */
end // System
```

1. The same as the current object – same as this
2. The same as some entity – same as entity_name

Type kinds: Multi-type (ADT *)

```
var v: Integer | Real is Integer.5
v := v + 5.5

FileOpen (fn: String): (File | Error) do
  if … do
    return FileDerivedType (…)
  else
    return ErrorDerivedType (…)
  end
end

fo is FileOpen (SomePath)
if fo is
   File: // Process fo as File type entity
   Error: // Process fo as Error type entity
end
```

Type kinds: Tuple type (ADT +)

```
t: (Integer, Real, String) is (5, 5.5, "Str")

SE (a, b, c: Real): (x1, x2: Real)
require a /= 0
do
   d is b*b-4*a*c
   if d >= 0 do
      return ((b+Math.sqrt(d)/2/a, (b-
Math.sqrt(d))/2/a)
   else
      …
   end
end

anArray: Array [Integer] is (10, 12, 33)
anArray(1) := anArray (3)
```

Type kinds: Range type

```
v1: 1..6 is 3
v2: 1 | 3 | 5 is 7 // compile-time error
v3: 1 | 3 | 5 .. 17 is 7
var v3: co1 | co2 | co3 .. co12 is co7
v3 := co13 // compile-time error
```

1. Range is a combination of constant objects of some unit-based type
2. If the unit-based type has declared constant objects in it range types may be constructed and their usage checked at compile time

Type kinds: routine(function) type

Nothing new – routines are 1st class citizens…

```
v1: rtn is rtn do … end

v2: rtn (T1, T2): T3 is rtn(p1: U1; p2: U2): U3
do … end
```

Type kinds: unit

Types (units) are 1st class citizens…

```
Type: unit … end /* entity called Type has the unit-type */
attr: Type /* attr has type -> Type*/

Type1: unit
  function (T1): T2
  procedure (T3)
end is LoadTypeDescriptionFromFile (…)

attr1 is Type1
attr1.procedure (new T3)
x is attr1.function (new T1)
```

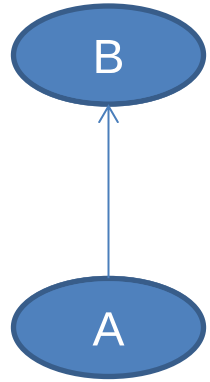
Type kinds: Detachable type

It is not a unique type kind it is a kind of an entity: entity can be of the always attached to the object kind and potentially not attached to the object kind

```
alwaysAttachedEntity: Type
detachableEntity: ? Type
detachableEntity := alwaysAttachedEntity // OK
alwaysAttachedEntity := detachableEntity
// Compile-time error
if detachableEntity is Type do
   alwaysAttachedEntity := detachableEntity
   /* OK as dynamic type of detachableEntity
conforms to Type */
   ? detachableEntity // detach
end
```

Compatibility = conformance + convertibility

1. Unit A conform to unit B if there is a path in inheritance graph from A to B
2. Signature foo conforms to signature goo if every type of signature foo conforms to corresponding type of signature goo




```
unit B
end

unit A extend B
end
```

goo (T_1, T_2, … T_n)

foo (U_1, U_2, … U_n)

if for $_i$ in $_1$ .. $_N$
U_i conforms to T_i

Compatibility = conformance + convertibility

```
val unit Integer [BitsNumber: Integer] …
  override := (other: Real) do … end
  override := (other: String) do … end
  := (): Real do … end
  := (): String do … end
  …
end
```

1. From-conversion => := procedure with 1 parameter
2. To-conversion => := function

```
i: Integer
i :=  5.5
i := “a string”
foo_real (i)
foo_string (i)
```

Type test - Duck typing

```
unit Duck // It can fly
    fly do StandardIO.print("Duck is flying") end
end

unit Sparrow // It flies too
    fly do StandardIO.print("Sparrow is flying") end
end

unit Whale // It does not fly but swims
    swim do StandardIO.print("Whale is swimming") end
end

while animal in (new Duck(), new Sparrow(), new Whale()) do
    /* Now it is necessary to check if the object ‘animal’
conforms to the type which is described as the anonymous unit-
based type which has only one routine – fly with no arguments.
Routines are specified using their signatures only */
  if animal is unit fly () end do
    animal.fly
  end
end
```

Summary

Everything for the language is defined using the language

O and I are 2 atoms

7+1 kinds of types (complete, sound, expressive, readable, etc. bla-bla-bla ☺)

Type compatibility = conformance + convertibility

Duck typing in place

THANK YOU VERY MUCH!!!