A unified type system for the modern general-purpose programming language focusing on reliability and verifiability

Abstract

The paper presents an overview of the type system which supports the convergence of procedural, object-oriented, functional, and concurrent programming paradigms relying on static type checking with smart type inference support and the ability to ensure dynamic type safety as well.

Keywords

Object, type, unit, class, module, interface, conformance, compatibility, type conversions, setters, reference and value objects, immutability.

CCS → Software and its engineering → Software notations and tools → General programming languages → Language types → Multi-paradigm programming languages

1. INTRODUCTION

The type system sets the basis for the reliable programming language and allows programmers to effectively express software design solutions using the power of the particular programming language raising the productivity of the software development process.

The modern tendency of convergence of different programming paradigms (merging procedural programming, structured programming, object-oriented programming, functional programming, and concurrent programming) forces the type system to support this.

In this paper, a highly condensed overview of the type system is presented and a programming language called SLang is used for the illustration of concepts. Necessary syntax constructs will be presented using simple notation based on conventions, where [term] means optional, {term} may be repeated zero or more times, term1 | term2 is the selection of term1 or term2, **bold font** is used to highlight keyword or special symbols.

Next is to define the notion of type as an important characteristic of every object during execution time (runtime). The type fixes the number of operations and their properties (signatures) as well as the size of memory required to store the object (number, valid values, and types of object attributes). So, a type is an abstraction used to describe the structure and behavior of objects.

Authors rely on concepts that are well-known by a broad audience of programmers and terms like class or variable will be used without formal definitions. Some definitions will be given right now to simplify the understanding of examples.

The unit is a named set of members, where a member can be a routine or an attribute. Routines stand for actions while attributes stand for data. If a routine returns some value as a result of its execution we call it a function otherwise a procedure. If an attribute can change its value during the program execution we call it a variable attribute (or simply variable) otherwise we call it a constant attribute (or simply constant or immutable attribute). Unit is very similar to class and the difference is that the unit incorporates characteristics of classes and modules (The term module is used like it was introduced in Ada (package) [2], Modula-2 (module) [4] – a generally available collection of data and routines with initialization) in one concept and the foundation for types.

So, the most important type is the unit-based type, and let's review units first.

2. UNITS

Any unit is a named collection of attributes or members. Such a definition sets away routines because they can be treated as constant attributes of routine type initialized with the routine signature and body. Units have other characteristics related to inheritance and usage; they will be explored below.

Every unit defines a type, and the name of the unit will be used as a type name. Such type is a unit-based type. The formal definition of the unit is

```
UnitDeclaration:
[final] [ref|val|concurrent|abstract|extend]
unit Identifier [AliasName] [FormalGenerics]
        [InheritDirective] [UseDirective]
{
        MemberSelection |
        InheritedMemberOverriding |
        InitProcedureInheritance |
        ConstObjectsDeclaration |
        MemberDeclaration
}
[InvariantBlock]
end
```

Unit is a central component and has a lot of elements. For the purpose of the paper, only `ConstObjectsDeclaration` and `MemberDeclaration` will be reviewed.

Specifiers indicate some characteristics of the unit and objects which can be built based on this unit-based type.

As a unit may inherit members from other units **final** specifier prevents further inheritance from this unit.

ref | **val** specifies the default form of objects which will be created using this unit as a type. The example below explains the difference. All objects of type Integer are to be values but not references to the integer number.

```
val unit Integer ... end
i: Integer is 5
```

Here, `i` is a value object. **is** works as a combination of entity declaration with initialization.

```
ir: ref Integer is 5
```

Here, `ir` is a reference object.

The default kind of object is a reference one. It's important to note that **ref** | **val** specifiers apply both to units and for particular objects and attributes. The unit-based type itself is not related to the form of objects of this type.

The **concurrent** specifier indicates that objects of this unit will be processed (executed) by a processing element that is different from the one which is used for all objects which are not marked as concurrent. The processing element is a general term for a physical processor, thread, process, remote server, or whatever computing machine. The mapping between the concurrent unit and actual physical executors is to be done outside of the programming language and it is not described here.

```
concurrent unit Philosopher
        // There are 5 of them
        // eating spaghetti...
end
```

If we like to ensure that there will be no objects created for the unit, it is to be marked as **abstract**. Of course, if there are some abstract routines within the body of the unit it is not possible to create an object of this unit type. So, it is not mandatory to mark such units as abstract as the compiler knows this, but if one likes to prevent objects creation for some units with having all routines as non-abstract then marking the unit abstract will allow to make it. Example:

```
abstract unit AnArray[G]
```

The **extend** specifier allows to extend already compiled unit with new members. For example:

Source #1 has

```
unit A
        foo do ... end
end
```

Source #2 has

```
extend unit A
        goo do ... end
end
```

Source #3 has

```
a is new A
a.foo
a.goo
```

Here, the second call to routine goo is valid if and only if the `A` unit extension was provided. In other words, sources #1, #2, and #3 will be compiled separately, but a compilation of Source #2 relies on the interface from Source #1, and a compilation of #Source 3 relies on interfaces of #1 and #2 sources.

As in many other OO-languages, **final** will not work together with **abstract** as it is out of sense to create a unit when it is not possible to create objects of this unit, and unit descendants are prohibited as well.

After the unit name aliasing name can be put (`AliasName`). It can be used to give an alternative name for the unit-based type. Some programmers do not like Integer they prefer **int** or `INTEGER`

```
val unit Integer alias Int
```

As we follow the style guideline that unit names should start with the capital letter.

Aliasing is a part of the type system. Although it does not create a new type, it allows to create unique names, and use short names instead of long ones. So, use with as aliasing can be put at the global level of the source like in the following example:

```
use StandardInputOutput as IO

IO.print ("Hello world!\n")
```

However, the name `StandardInputOutput` still stays as a valid name of the unit. So, unit-based types `StandardInputOutput` and `IO` refer to the same type.

`FormalGenerics` is an optional parametrization of the unit with some unit-based type, or value, or routine. For such kind of parametrization, the term *genericity* is used. The notation uses square brackets.

```
abstract unit AnArray[G]
```

where `G` is the name of the type which is to be provided to get a particular instantiation of the unit-based type.

```
abstract unit OneDimentionalArray
                    [G extend Any init()]
```

`G` can be constrained meaning that any type which is used for instantiation is to be conformant to the type specified as

a constraint. In the case of the example above it should be a descendant of Any. If it's necessary to create objects of the formal generic type we need to know which initialization procedure (constructor) to be used – in this example requirement for the instantiating type is to have an initialization procedure without arguments.

```
unit Array [G extend Any init(),N: Integer]
      extend OneDimentionalArray[G]
```

Here we have two generic parameters G and N and the second one (N) is the constant of the type which is specified (Integer).

`InheritDirective` specifies from which units this unit inherits members. Here it is essential just to mention that inheritance is multiple and does not use the subobject concept. Every unit member is inherited on its own. The keyword **extend** (which is well-known by many programmers) is used to highlight the set of parent (base) units. The example above in the section on generics shows that unit Array inherits all members from the unit OneDimentionalArray.

`UseDirective`. The idea of a module as a container of functionality seems to be similar to that of [1]. However, there are some other differences between classes and modules. The key point is that based on the class one may create an unlimited number of objects while for the module there will be just one object created and properly initialized. Modules are created and initialized implicitly while object creation is a special statement or expression. So, it implies that a unit may be used as a module if and only if it has no initialization procedure or at least one initialization procedure with no arguments. The example below highlights that

```
use StandardInputOutput as IO
IO.print("Hello world!\n")
```

Here, IO is the name of the module which is created and initialized at some moment of the program execution (actually, two options are possible – to create all module objects at the program start or right before the first access to the module members).

```
io is new IO.init(IO.TextMode)
```

Here, io is an object which is initialized with the creation of a new object of type IO

```
io1 is new IO.init(IO.GraphicalMode)
```

An unlimited number of objects like io1 can be created, initialized, and used when uint is used as a type.

```
io.print("Hello world!\n")
```

In this example, IO is a global module which is available across all components of the program, but if we like to have a module dedicated to the unit hierarchy (current unit and all its descendants (derived units)) then we can specify it using `UseDirective` syntax like this

```
unit A use B ... end
```

So, inside of A all calls of the form B.foo() are calls to the functionality fo the module B

If access to the global unit B is required then it is possible to give a local name for the B which is used as a module for A unit hierarchy like this

```
unit A use B as BB ... end
```

So, inside of A all calls of the form B.foo() are calls to the functionality of the global module B, and calls like BB.foo() are calls to the local module routine

Next is the `MemberDeclaration` section of the unit declaration

2.1 UNIT MEMBERS

There are 3 kinds of unit members – unit routines (procedures or function), unit attributes (data fields), and unit initialization procedures. By default, all unit members are visible for unit descendants and clients and this visibility implies an ability to call routines and read the attributes while clients are not able to change the value of attributes and override routines. Of course, there should be a mechanism to change the visibility of the particular unit member or a group of members. One may limit visibility in the following ways

```
unit A
      rtn1 do end
      // Routine 'rtn1' is visible for all
      // descendants and clients

      {} rtn2: T do end
      // Routine 'rtn2' is visible for all
      // descendants only

      {this} rtn3 do end
      // Routine 'rtn3' is visible only for
      // the current unit A

      {B, C} rtn4 do end
      // Routine 'rtn4' is visible for all
      // descendants and clients B and C

      {}:    // Group of members with the
             // same visibility

             attr1: T1
             var attr2: T2

      end
end
```

One may notice that the second attribute is marked with the **var** specifier while the first one has nothing. By default, all attributes are in fact constants with initialization. So adding var, it will be possible to change the value of this attribute and its content at any time during program execution. The concept of constantness (immutability) will be explored later but now let's review initialization procedures.

2.2 UNIT INITIALIZATION PROCEDURES

When an object is being created there should be a way to put it into a consistent stage that fully matches its invariant. That is why an initialization procedure is needed (a

constructor or a creation procedure in other programming languages) as the only task it has is to initialize all attributes of the unit. The straightforward choice for the name was "init" and as the name of the initialization procedure is known it can be skipped when a new object is being created, as well the empty parenthesis if init has not arguments. So, here is a reduced example of the initialization procedure of unit Boolean

```
val unit Boolean
    init do
        data := 0xb
    end

    {} var data:
            Bit[Platform.BooleanBitsCount]
end // Boolean
```

Variable attribute `data` that is not visible to the clients of `Boolean` is initialized with zero, interpreted as false. So, here is implicit magic (no defaults) – all units including basic ones explicitly define initial values for all their attributes.

```
b is new Boolean
```

This means that object `b` is created with the value false. This is a short cut for a declaration like this

```
b: Boolean is new Boolean.init()
```

A unit may have several init procedures and the programmer can select the one which is required for the particular case.

```
unit A
    init (a1: T1; a2: T2) do end
    {} init (a: A) do end
    foo do
        a is new A(this)
    end
end
```

In this example, `a` is a local attribute of routine `foo`, created by **`new`** and initialized with the second **`init`** procedure which is available only for this unit.

```
a1 is new A.init(new T1, new T2)
a2 is new A(new T1, new T2)
```

As **`init`** name is known it can be skipped while creating new objects. Outside of unit A only one initialization procedure is visible and has to be used while creating new objects.

2.3 UNIT INVARIANTS

Unit invariant is a set of predicates that state when objects of this unit type and its descendants be consistent. It is a requirement to objects consistency – that is why the keyword **`require`** is being used to highlight that.

```
abstract unit Numeric
       one: as this abstract
       zero: as this abstract
       // Declarations of * and +
       // are skipped
require
       this = this * one
       zero = this * zero
       this = this + zero
end // Numeric
```

Every numeric object of a type that is a descendant of `Numeric` should implement concepts of `one` (1) and `zero` (0) and should be consistent with the invariant stated in `Numeric`. So, if some operation is applied to an object of some type then after completing the operation the unit invariant is to be checked to ensure that object is still in the consistent state and ready again to perform new operations.

2.4 UNIT SETTERS AND GETTERS

As all visible unit attributes are directly accessible for clients and descendants – their names are effective getters. For setters, it is rather convenient to use syntax like `a.b := expr` instead of `a.b.set_b(expr)`, but semantically they have the same meaning – we need to call some procedure that will set the value of some unit attribute to a proper state. So, the straightforward approach is to use `:=` as the name of the setter and associate it with the attribute declaration.

```
unit A
    var attr1: T1 :=
         alias setAttr1 (other: T2) do...end
end
```

In unit `A`, the variable `attr1` has a setter with an argument of type `T2` and this setter has an additional name `setAttr1`.

After that, objects can be defined and setter used. Both last statements do the same – they set attribute `a` to the same value.

```
a is new A
a.attr1 := new T2
a.setAttr1 (new T2)
```

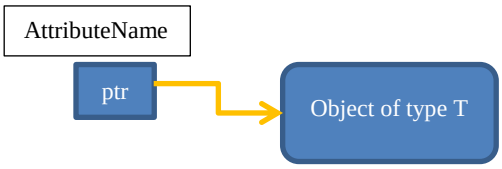
2.5 IMMUTABILITY

As `a: Type` is a declaration of a constant attribute, a similar scheme is applied for routine arguments. It implies that it is not possible to assign new values to formal arguments. Other implications of the constantness status of an attribute that it is not possible to change the state of an object. It implies that any calls to routines that change such a state are statically detected by the compiler and a proper error message is generated. So, if an attribute is marked as **`var`** attribute – assignment to this attribute and any correct routine call will be a valid action. If no mark in place or attribute is marked as **`rigid`**, then the attribute can only be initialized once, and then it will keep its value. In the case of **`rigid`**, the whole object tree accessible from this object is immutable. So, `rigid` implies deep constantness of an attribute while no mark means shallow constantness.

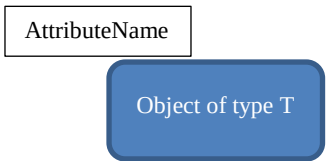
As data attributes can be of two kinds – reference and value, the semantic of the assignment statement is a bit different. There are four possible cases

```
ref1 := ref2
// Copy ref2 into ref1.
// After the assignment, they both point
// to the same object.
```

```
val1 := val2
// Field by field copy of the object named
// val2 into the corresponding fields of the
// object named val1.

ref := val
// Clone the object named val and reference
// to this clone is put into ref.

val := ref
// Field by field copy all fields of the
// object pointed by ref into the
// corresponding fields in the object named
// val.
```




```
AttributeName: ref T
```




```
AttributeName: val T
```

Once again: the type itself is agnostic to the kind of objects which will be created. So, **ref** and **val** objects of the same type can be easily assigned to each other (boxing unboxing is done by the compiler automatically). The example below illustrates this.

```
unit A
   var attr: Type := (other: Type) do
                          attr := other
                      end

   foo (arg: Type) do
     // The assignment below generates
     // a compile-time error as 'arg'
     // is a constant object.
     arg.attr := Type
   end

   goo (var arg: Type) do
     arg.attr := Type
     // This assignment is OK, as 'arg' was
     // explicitly marked as mutable.
   end

   // The immutable attribute should not
   // have a setter. The code below leads to
   // a compile-time error.
   attr2: T1 := (other: T1) do ... end
end // A
```

One more illustration of how **var** works in the context of **ref** and **val** objects.

```
i is 6
```

Type of `i` is deduced by the compiler based on the type of constant object 6 into **val** Integer.

```
ir: ref Integer is 6
```

Here, `ir` has got an explicit type and 6 will be cloned into **ref** `Integer`. No operations that change the contents of the object can be performed over `i` and `ir` – they are immutable. Compile-time errors will be raised for both following statements.

```
ir++
i++
```

The following code compiles and run with no issues. `++` is the routine of unit `Integer`.

```
var j is 5
var jr: ref Integer is 5
j++
jr++
```

So, **ref** and **val** kinds of objects are completely unrelated to the immutability status of objects and both mechanisms give the full control over objects' semantic. Now we have described how to define immutable attributes but how can we properly define constants like numbers, characters, string, and value constants of any type. This leads to the constant objects section.

3. CONSTANT OBJECTS

3.1 BACKBONE - TWO FUNDAMENTAL CONSTANTS

Learning computer science usually starts with two simple idioms – 0 and 1 (zero and one). Generalizing we may state that we have two signs circle and bar and start defining everything in the digital world combining these signs into sequences and giving a different interpretation of such chains. Binary digit (bit) was selected as a term to represent this. So, we have defined some unit Bit which has two constant objects of type `Bit`: `Bit.Ob0` and `Bit.Ob1`. An example with the part of the source code of unit `Bit` illustrates how these constants are defined.

```
val unit Bit
  const Ob0, Ob1
    // As unit Bit has no init procedure,
    // Ob0 and Ob1 are just two different
    // objects, and Ob0 and Ob1 are their
    // names and values at the same time.
  end

  // Function & is fully defined in the
  // source code of the unit. Both names
  // & and 'and' can be used.
  pure & alias and (other: as this): as this
       => if this = Ob0 do Ob0
          elsif other = Ob0 do Ob0 else Ob1

end // Bit
```

All other types are based on unit `Bit`. And has explicit source code with the implementation of all routines (methods or member-functions). No more need to remember what functions can be applied to 'int' – there is a source code and as we inspect any other type basic types can be inspected too.

3.2 BASIC UNITS – BASIC TYPES

Using the same approach all basic types are being introduced. As one more example, we will use some fragments of units `Integer` and `Integer[BitsNumber:Integer]`. It illustrates one more concept of unit names overloading which works well within our type system.

```
val unit Integer
   extend Integer[Platform.IntegerBitsCount]
   ...
end
```

That is a general `Integer` that uses the platform description constant, the number of bits in integer for setup

```
val unit Integer[BitsNumber: Integer]

  // Thus, we can instantiate this type like
  // Integer[4] or Integer [16] when we need
  // particular types of a particular size
  // in bits

  minInteger is - (2 ^ (BitsNumber-1))
  maxInteger is 2 ^ (BitsNumber-1)-1

  // This is an ordered set defined as a
  // range of all integer constant values
  // (objects)

  const
      minInteger..maxInteger
  end

  ...

  init do
     data := new Bit[BitsNumber]
  end

  {} data: Bit[BitsNumber]

end
```

For types like `String` and `Bit[N]` regular expressions are being used to define all possible constants of these types.

3.3 CONSTANT OBJECTS – THE GENERAL CASE

Every unit may define all known constant objects or specify the rule with help of regular expression how all constants will be generated. Block **`const ... end`** aims to do that.

For example, `Integer.1` is a valid constant object of type Integer.

To skip unit name prefix apply **`use ... const`** – import all constants into the place where one needs them.

As an example of constants import, we may consider unit Any which resides at the top of all units (like class `Object` in Java)

```
unit Any
  use const Integer, Real, Boolean,
            Character, String,
            Bit[2**Integer.MaxInteger]
```

All constant objects from basic units are imported into `Any` and respectively to all other units allowing usage of these constants without respective unit name prefix.

Below is an example of weekdays that shows that constant objects replace enumeration types.

```
unit WeekDay
  const
    Monday, Tuesday, Wednesday, Thursday,
    Friday, Saturday, Sunday
  end
end

use const WeekDay
```

This imports all constant from unit `WeekDay` into this script code. First, call procedure `foo` with the parameter `Monday`.

```
foo(Monday)
```

Here is the declaration of `foo`. It contains an example of pattern matching inside.

```
foo (day: WeekDay) do
  if day is
    Monday .. Friday:
      StandardIO.print("Go to the office")
    Saturday, Sunday:
      StandardIO.print("Do what you like!")
   end
end
```

The last synthetic example shows the exact meaning of constant objects. Some unit `A` is declared. It defines three constant objects and uses all three initialization procedures for their creation. After the unit code, the small script shows how type `A` can be used.

```
unit A
  const
    a1.init,
    a2.init (new T),
    a3.init (new T1, new T2)
  end

  init do end
  init(arg: T) do end
  init(arg1: T1; arg2: T2) do end
end

x is A.a1
```

Here, `x` is defined as a valid constant object and initialized with the value of the constant object from `A`.

```
var y is A.a2
```

However, the attempt to declare a variable and initialize it with the const object will lead to a compile-time error.

4. TYPES

As mentioned before, there are 8 kinds of proposed types – unit-based type, anchored type, multi-type, detachable type, tuple type, range type, routine type, and unit-dependent type. Every type has an explicit description – type declaration.

4.1 UNIT-BASED TYPES

Unit-based type is the most commonly used kind of type. Every new unit declaration defines a new type. Such unit declaration explicitly defines all attributes and all routines of this unit – fixing the set of operations over objects of this type and size of objects of this type in memory. Units are a more general form of classes and modules. Units may inherit like classes and may be used like modules (provide a single object, supplier of functionality). All examples above used unit-based types.

4.2 ANCHORED TYPES

Anchored type is the type, which is the same as another entity has. Such kind of types was introduced in Eiffel [3]. It works as an automatic overriding while inheriting and allows not to repeat the exact type name. Example

```
b: as a
```

So, `b` is defined as having type the same as `a` has.

```
x: as this
```

Here, `x` has a type similar to the current unit. In descendants type of `x` automatically changes.

4.3 MULTI_TYPES

Multi-type states that objects of this type can be one of the types specified in the type declaration. So, the set of operations which can be applied to such objects is an intersection of operation from all types included in the multi-type declaration. So, it allows producing code, which works with objects of already compiled units with no need for inheritance. In the example below, `c` may be assigned with objects of types `A` or `B`.

```
c: A | B
c := new A
c := new B
c.foo(expression)
```

Both types `A` and `B` must have a routine `foo` with the proper signature for the *`expression`* to be compatible with both signatures. The exact definition of type compatibility will be given later.

4.4 DETACHABLE TYPES

Detachable type in the form of `?UnitBasedType` allows to declare attributes with no initial value and such attributes can be initialized later with objects of `UnitBasedType` or its descendants and dynamic type check has to be applied to deal with such objects (call member-routines or read member-attributes). Example

```
d: ?A
if d is A do
   d.foo
   ?d
end
```

`d` is declared as having no value. So, `d` cannot be used unless its type is checked at runtime. Inside of the do block (then part) of the if statement `d` has the type of `A` till the first assignment to it or detachment `?d`.

4.5 TUPLE TYPES

Tuple type defines a group of entities of potentially different types specified in the type declaration. The number of entities is part of the type declaration. It is possible to name these tuple fields with identifiers for access by name. The example below introduces `e` as a group of values. Its type is a tuple with three types in the specified order and `e` is initialized with tuple value.

```
e: (Integer, Real, String) is
              (5, 6.6, “Hello world!”)
```

Next is the square equation solution, which uses tuple to get the result. Type of object `(x1, x2)` is `(Real, Real)`. Function `SolveSquareEquation` returns a tuple in which has named fields in it. Both ways to call it are presented below.

```
SolveSquareEquation (a, b, c: Real):
      (r1: Real; r2: Real) do ... end

(x1, x2) is SolveSquareEquation(1.0,2.0,3.0)
roots is SolveSquareEquation (3.0, 2.0, 4.0)

x1 is roots.r1 // First root
x2 is roots.r2 // Second root
```

Important comment: array is a kind of tuple when all elements have the same static type. That is another reason why access to array elements uses the syntax similar to access to tuple elements by index.

4.6 RANGE TYPES

The range type explicitly defines a set of possible values objects of this type may have. There are two kinds of this type presented below.

```
f: 1..6
```

`f` can store Integer values between 1 and 6.

```
g: 1|3|5|7
```

`g` can have odd integer values between 1 and 7. `f` and `g` have different types, so any attempts to assign will lead to compile-time errors. Both assignments below are wrong.

```
f := g
g := f
```

4.7 ROUTINE TYPES

The routine type defines objects which are routines and it means that activation (call or application) of the routine associated with the object can be done later. Routines are treated as first-class citizens. The example below defines procedure `foo`, which can be called with the routine object which has the routine type – a function with 2 arguments of types `Type1` and `Type2` returning objects of type `Type3`.

The body of foo contains a call to routine passed as an argument.

```
foo(h: rtn (Type1, Type2): Type3) do
  x is h (new Type1, new Type2)
end
```

foo can be called providing the inline routine object.

```
foo(rtn (Type1; Type2): Type3 do
            return new Type3 end)
```

4.8 UNIT-DEPENDENT TYPES

The unit-dependent type defines objects which define types as first-class citizens. One can declare an attribute of type unit and provide a full description of this unit at some time and then use the name of this attribute as a type for declaration of other entities.

```
Type0 is new unit
              foo do end
              init do end
              var attr: X
            end
```

Attribute Type0 has a type equal to the unit type deduced by the compiler. This unit type is characterized by members: routine foo, initialization procedure, and a mutable attribute attr.

```
Type1: unit is unit
                foo do ... end
              end
```

Attribute Type1 is defined as having type unit initialized by inline unit declaration. Also, it is possible to specify the unit interface of interest and then dynamically assign conforming types to this variable. The order of unit members is not essential; that is the difference from tuples.

```
Type2: unit
         f1: T1
         f2: T2
         r1(T1,T2)
         r2(T1): T2
         init()
       end is new unit
              r1(T1, T2) do end
              r2(T1): T2 do end
              init() do end
            end
```

Here, the type of Type2 is limited with some interface specified as unit type. So any type which conforms to the interface can be assigned to Type2. The initialization part should not repeat the attributes specified in the type description, but new ones may be added and all routines should get their bodies.

```
Type3: ?unit foo(), init() end
```

Type3 attribute is not initialized but we know its interface. Now we can use new types for ordinary attributes declarations.

```
a0 is new Type0.init()
a0.foo
a1 is new Type1
a1.foo
a2 is new Type2.init()
a2.r1(new T1, new T2)
a3: Type3 is new Type0.init ()
a3.foo
```

What else can be done with attributes of the unit type? By default, assignment works for them and they can be used for declarations. Of course, conformance rules are to be adjusted for such types. But it is possible to build such a unit type during the program execution like as follows:

```
Type4 is new unit end
Type4.add(rtn foo () do end, var x: Integer)
Type4.add (y: Real; init do end)
```

As we have no exact static information about the nature of Type4, we have to dynamically test it. If it has proper init procedure or require routine with necessary signature and then call.

```
if Type4 is unit init () end do
    a4 is new Type4.init ()
    if a4 is unit foo () end do
        a4.foo ()
    end
end
```

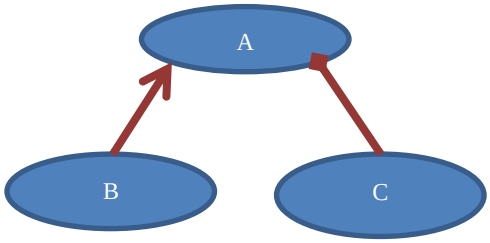
Among basic units, there is a special one that defines the unit type. The code of procedure add shows how it is possible to deal with an unlimited number of arguments.

```
unit unit // Name of the unit is unit!
     add (members: ()) do
      while member in members do
      Runtime.addMemberToUnit (this, member)
      end
     end
end
```

One more aspect of such types is using them within the generics approach. Instead of parametrization by a constant of an enumerated type, one can provide an expression. See an example below

```
var v1 is new Array[String, 5]
```

v1 will be an array of strings with five elements properly initialized by Array init procedure.




```
var v2 is new Array[String, 6]
```

v2 will be an array of strings with six elements properly initialized by the Array init procedure.

```
v1 := v2
v2 := v1
```

Both assignments are valid as v1 and v2 have the same type Array[String; N: Integer].

```
var v3 is
  new Array[String,StandardIO.readInteger()]
```

The actual type of generic instantiation attribute `v3` will be identified during execution.

```
v1 := v3
v3 := v2
```

However, both assignments are valid as `v1` and `v2` have the same type `Array[String; N: Integer]`. So, type compatibility is very essential.

4.9 TYPE COMPATIBILITY

It is essential to define well when assignments are valid and when overriding is valid while inheriting. The latter is described by the signature conformance while the assignment is driven by the following rule. The type of the expression on the right side of the assignment should either conform to the type of the writable on the left side or have a proper conversion routine in place. So, type A is compatible with type B if A conforms to B or objects of type A can be converted into the objects of type B. Pictures below will use the legend that every oval denotes a unit and every arrow means 'inherits from' aligned with the direction of the arrow. Rombus-ended edge means inheritance with no conformance (not able to make polymorphic assignments)

4.9.1 TYPE CONFORMANCE

The simplest case of conformance is that each type conforms to itself.

```
a: A is new A
```

Unit conformance is based on the idea to check if there is a path in the inheritance graph between the current unit type and another one. And this path should consist only of conformant inheritance edges.

```
unit A end
unit B extends A end
```

That is a conformant inheritance.

```
unit C extend ~A end
```

That is a non-conformant inheritance.

```
a: A is new B
```

Valid as `B` conforms to `A`.

```
a: A is new C
```

Not valid as `C` does not conform to `A`.

When a type is a generic instantiation then in addition to unit type conformance it is necessary to take into account type by type conformance of all elements of the instantiation. Notice that square brackets are used to highlight generics. Access to tuples and arrays is done using parentheses as these are function calls with parameters.

```
unit A[U, V] end
unit B[X, Y] extend A [X, Y] end

unit T1 end
unit T2 end
unit S1 extend T1 end

unit A[A, B, C] end
a: A[T1, T2] is new A [T1, T2]
```

Valid as types are identical.

```
a: A[T1, T2] is new A [S1, T2]
```

Valid as `S1` conforms to `T1`.

```
a: A[T1, T2] is new A [T1, S1]
```

Not valid as `S1` does not conform to `T2`.

```
a: A[T1, T2] is new B [T1, T2]
```

Valid as `B` conforms to `A` and has identical instantiation types.

```
a: A[T1, T2] is new B [S1, T2]
```

Valid as `B` conforms to `A` and has conformant instantiation types.

```
a: A[T1, T2] is new B [T1, S1]
```

Not valid as `S1` does not conform to `T2`.

```
a: A[T1, T2] is new A [T1, T2, S1]
```

Not valid as `A` with 3 generic parameters does not conform to `A` with 2 generic parameters.

Tuple conformance. All tuples are of the same type – tuple type. It means that we need to consider (similar to generic instantiations) by-element conformance of element types.

```
a: (T1, T2) is (new T1, new T2)
```

Valid as types are identical.

```
a: (T1, T2) is (new S1, new T2)
```

Valid as `S1` conforms to `T1`.

```
a: (T1, T2) is (new T1, new S1)
```

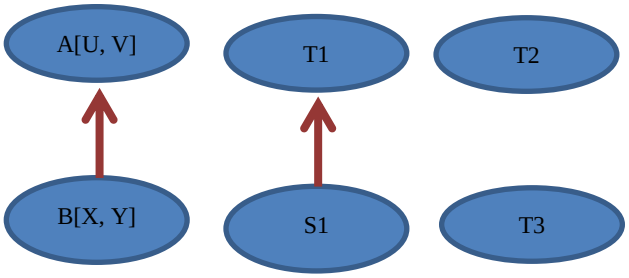
Not valid as `S1` does not conform to `T2`.

```
a: (T1, T2) is (new S1, new T2, new S1)
```

Valid as all elements of the longer tuple, which has corresponding elements in the shorter one, conform to them.

Last but not least is **unit type conformance**. All unit types are of the same type – 'unit', similar to tuple conformance. So, we need to look at a member after a member to check if they conform to each other. The difference from tuples that tuples have an order of elements in the tuple but unit types do not. But every member of the unit type has a name. And search by name identifies the subset of members which will define the conformance. So, if we have two unit types A and B then A conforms to B if for every member of A there is a member with the same name in B and its signature in A conforms to the signature of the corresponding member in B and B has not other members. Common sense logic brings the idea that to an empty unit any unit type will conform. Any 'thinner' unit type will always accommodate in terms of conformance the 'thicker' one. Empty unit means any unit!

```
var A is unit end

var B is unit
  foo (T1, T2): T3
  goo (T3)
  var attr: T1 := (T1)
  // It has a setter with an argument
  // of type T1
end

var C is unit
           foo (S1, T2): T3
           goo (T3)
      end

var D is unit
           foo (S1, T2): T3
           goo (T3)
           var attr: T1 := (S1)
           // it has setter with an argument
           // of type S1
           too (T1, T2, T3)
         end

A := B  // Valid as any type conforms
        // to an empty type

B := C  // Not valid as C lacks member
        // called attr

B:= D   // Valid as all D members fit all
        // B members in terms of conformance
        // and D has extra members – it is
        // thicker than B
```



4.9.2 TYPE CONVERTABILITY

Here, conversion routines are considered as they also play important roles in assignments. There are two types of conversion routines: from-conversions and to-conversions. The first one is a procedure with one parameter and the second one is a function with no arguments. Let's examine the following example with units `A` and `T`.

```
unit A
   := (other: T) do ... end
   // This is a from-conversion procedure,
   // which has some algorithm how to
   // perform a conversion from objects of
   // type T into the objects of the current
   // type A. T is just some empty type.

   := (): T do ... end
   // This is a to-conversion function that
   //  creates a proper object of type T
   // and works well for assignments too.

   foo (arg: T) do end
   // Procedure 'foo' will be used to show
   // how converters work
end

unit T end
```

At first, let's create a valid object of type `A`, and then different conversions will be done using an assignment statement.

```
var a is new A
a := new T
```

Here, `a` can be assigned with an object of type `T` as it has a from-converter procedure.

```
a.foo (new A)
```

This call is valid as well as unit `A` has a to-conversion function to type `T`.

Here is a brief review of routines' signature conformance which also has similarity with generic instantiation conformance and uses tuple conformance. If we have routine foo with signature `S1` and routine goo with signature `S2` then `S2` conforms to `S1` if they have the same number of elements and every type element of signature `S1` conforms to the appropriate element of signature `S1`. Let's consider the following example

```
unit A
    foo (T1; T2; T3): T4
end

unit B extend A
    override foo (U1; U2; U3): U4
end
```

In this example, the signature of `foo` from `A` is `((T1, T2, T3), T4)`, and `foo` from `B` has `((U1, U2, U3), U4)` and the task is equal to tuple conformance. Tuple `((U1, U2, U3), U4)` conforms to the tuple `((T1, T2, T3): T4)` as they have the same number of elements – two in this case (for the procedure we may just drop the return type) and for the first element we again have tuples conformance case - whether `(U1, U2, U3)` conforms to `(T1, T2, T3)` and check if `U4` conforms to `T4`.

Some notes about the name and structural type equivalence. Below is an example in Ada[2], which presents name equivalence – type `Integer_1` is not compatible with type `Integer_2` as they have different names! But structurally they are identical.

```
type Integer_1 is range 1 .. 10;
type Integer_2 is range 1 .. 10;
A : Integer_1 := 8;
B : Integer_2 := A; -- illegal!
```

We can choose between two different approaches. The first one is right below

```
a : 1 .. 10 is 8
b : 1 .. 10 is a
```

Here, `a` and `b` have the same type: range type `1..10` and `a` can be assigned to `b`.

In the second case when one likes to introduce new types, type `Integer_1` is different from `Integer_2` and they are not compatible.

```
unit Integer_1 extend Integer
require
    this in 1 .. 10
end

unit Integer_2 extend Integer
require
    this in 1 .. 10
end

var a is new Integer_1

var b: Integer_2 is a
```

Declaration of `b` leads to compile-time error as the type of `a` is not compatible with the type of `b`.

So, support of name equivalence is in place but the term name is treated a bit wider. `1..10` is treated as the type name, `A | B` is the type name too, and `(T1, T2, T3)` is also a type and its name is a tuple `(T1, T2, T3)`, type "**`as this`**" is compatible to the type of the unit where an attribute of such type was declared.

4.10 DUCK TYPING

The popular thing is duck typing. It also can be interpreted in terms of the conformance test. As an ability to fly means that we can imagine a hypothetical unit `Flyable` with one abstract procedure `fly` and check if the object of interest conforms to this unit-based type or not. The trick is that we do not need to enforce to change the inheritance graph for that. We need just to construct such a unit on the fly, keep it anonymous, and just apply the proper check. Let's consider the following example which is used for other programming languages

```
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 (Duck, Sparrow, 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
    // signature only.
  if animal is unit fly () end
  do
    animal.fly
  end
end
```

Here are a few caveats. What is the static type of animal to be determined by the type inference process? If units `Duck`, `Sparrow`, and `Whale` have the nearest common ancestor, this unit will be the type of animal. If such unit was not explicitly mentioned thru extend directives then Any will be such unit. So, the process terminates in any case. If there are several nearest common ancestors then the process can be run for them recursively.

5. CONCLUSION

The paper presents the uniform type system which supports the convergence of different models of programming, allows to have static typing with type inference, to have all types and values be explicitly and fully defined using the same programming language. For that, the concept of the unit is used and it is defined as a combination of class and module concepts. Types compatibility if fully and explicitly defined using type conformance and type conversion. Both conformance and conversions are fully defined too. The approach which allows treating manifest constants as immutable objects of the proper type is introduced, it works well for basic types and user-defined ones. It superceeds enumerations and sets the background to have the programming language which is fully defined using the language itself.

6. REFERENCES

[1] Clemens A. Szyperski: Import is Not Inheritance. Why We Need Both: Modules and Classes, ECOOP 1992.

[2] International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.

[3] Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice-Hall. ISBN 0-13-629155-4.

[4] International Standard: ISO/IEC 10514-2:1998 Information technology – Programming Languages – Modula-2.