We all use slang, let's program in SLang

Eugene Zouev,
Innopolis University

Alexey Kanatov,
Huawei

Agenda

- Introduction
- Compilation units – anonymous procedures and units
- Operators – if & loop, super block
- Approach to inheritance, feature call validity
- Null-safety and non-initialized attributes
- Constant objects
- Standard library basics
- Extended overloading
- Unit extensions
- Generics
- Dining philosophers
- Summary

Personal introduction

- 10+ years compiler development
- 15+ years of SW R&D management
- 4 years of teaching
- In **2010** started with design of E#
- **Huwaei**, Chief academic consultant
- **Innopolis University**, Associate pprofessor, head of the laboratory of Data analysis and finance technologies
- **Samsung**, Compiler, Platform, System AI Tools department manager
- **WorldQuant** Research (Eurasia), quantitative investment management company, branch director
- **Intel**, head of Intel Compiler QA, Intel Compiler Russia, Moscow Site, Intel Platform Simulator
- **Object Tools**, Inc. Senior Software Engineer, Visual Eiffel compiler, architect and key developer
- **‘Zenon’** Ltd., Software Engineer, databases, first Russian stock exchange software
- “Tsaritsyno” **Centre of education 548**, Informatics Teacher
- Cybernetics faculty, Moscow Engineering Physics Institute, **MEPhI**, Bachelor-Masters-Postgraduate, Ada compiler & Modula-2 tools

Initiative introduction

- **We met in 2014 and started cooperation**
- **Authors' background**: C++, Zonnon, Ada, Modula-2, Eiffel, … talk and battle ☺
- **Driving force** – looking for better programming language design is the driving force
- **Terminology**: feature (characteristic) – routine or attribute, attribute – variable or constant, routine – procedure or function; inheritance graph & conformance; module, type, class
- **Main task of the talk** is to give high-level overview of concepts features which could be of interest

Compilation units

3 kinds:

- **Anonymous procedure:** sequence of operators
- **Standalone-routine:** scope, formal parameters, pre & post conditions, body
- **Unit**: named set of routines, attributes, and invariant
  - Can be generic - type or constant expression of enumerated type parameterized
  - Unit defines a type
  - Unit supports inheritance
  - Unit support direct usage (acts as a module)

Unit(module) name

```
StandardIO.put("Hello world!\n")
aFunction (“ha-ha-ha”)
```

New shorter name of the unit

```
use StandardIO as io
aFunction(aString: String) do
    io.put("Test!\n")
    c is C("This is a string“)
    io.put(c.string + “ “ +
aString)
end
```

Standalone procedure

```
unit C
    string: String
    init (aString: as string) do
        string := aString
    end
end
```

Unit

Unit … Some important definitions …

- Unit – named set of attributes and routines
- Unit – named set of properties and functions
- Unit – named set of members (data members and member functions)

Is unit a type – yes! The key thing unit has explicit definitions of all its features (members)

Type is a more general concept

```
use B | C | D as A
aFunc (parameter: as this)…
unit Array [G] …
```

Type is characterized by set of values (data space) and set of operations
Type may have a name

- Type and unit are compile time entities
- Instances or objects are runtime entities
- Objects can be of reference or value nature

Dual syntax :-)

```
StandardIO.put("Hello world!\n")
routine (“ha-ha-ha”)

use StandardIO as io
routine(aString: String) {
  io.put("Test!\n")
  c is C<T>("This is a string“)
  io.put(c.string + “ “ +
aString)
}

unit C <G>
    string: String
    init (aString: as string) {
        string := aString
    }
}
```

```
StandardIO.put("Hello world!\n")
routine (“ha-ha-ha”)

use StandardIO as io
routine(aString: String) do
  io.put("Test!\n")
  c is C[T]("This is a string“)
  io.put(c.string + “ “ +
aString)
end

unit C [G]
    string: String
    init (aString: as string) do
        string := aString
    end
end
```

Syntax is just a form, one may select the one which suits better …

Units – 3 in 1 (class, module, type)

Usage (module)

Client gets access to visible features of the module

Inheritance (class)

Unit inherits features of the base units treating them as classes

Typification (type)

Each unit defines a type. This type can be used to define unit attribute, local or argument of routine

Usage(module)

```
StandardIO.put("Hello world!\n")
routine (C)
```

Inheritance(class)

```
unit C extend B, ~D use B
end
```

Typification (type)

```
routine(b: B) use D do
    D.foo
end
```

Usage(module)

```
unit B
    foo do
    end
end
```

Type is the universal and the most high-level concept

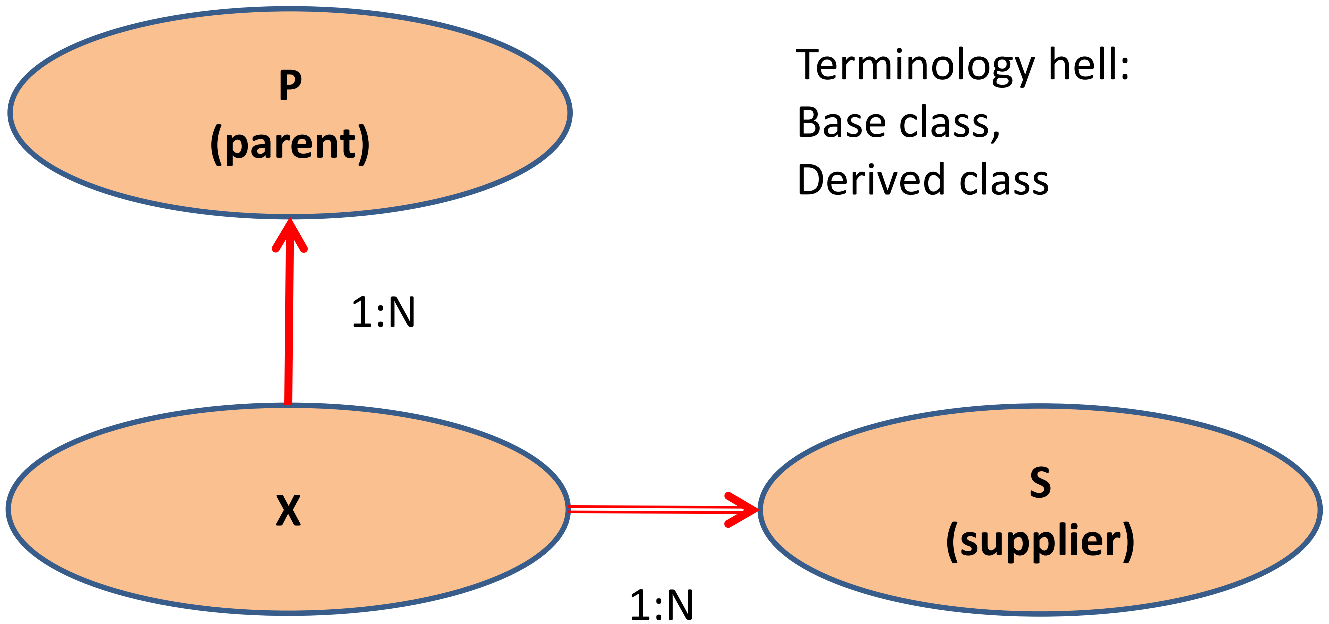
Relations between types/units

- Inheritance
- Usage

X inherits P

X uses S

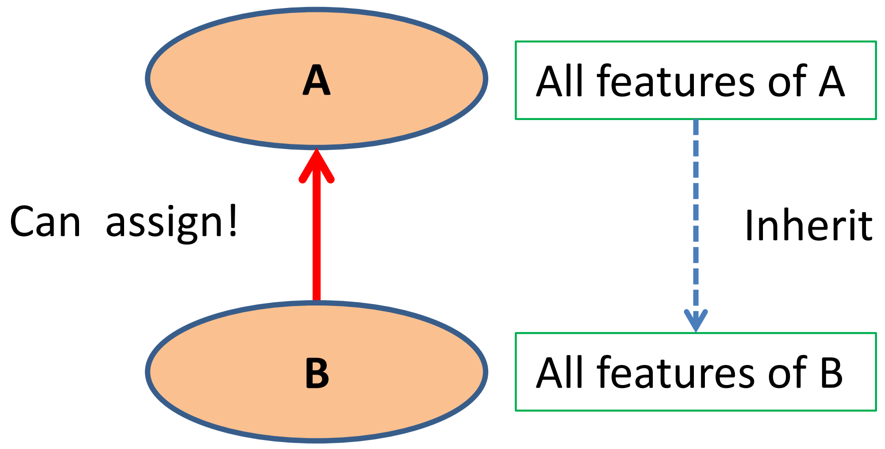
X is a client of S



Terminology hell:
Base class,
Derived class

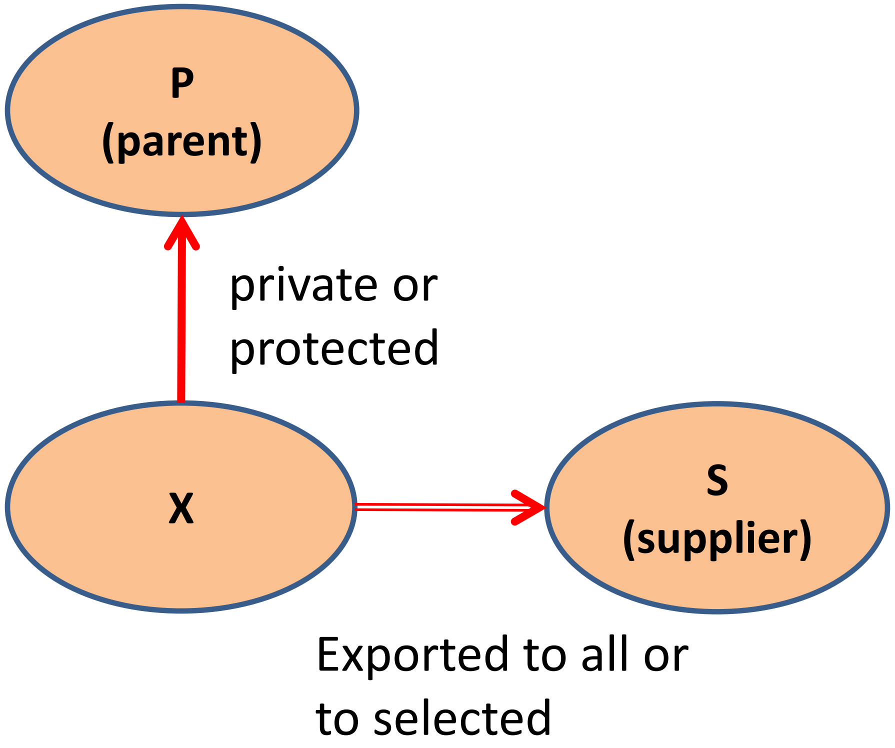
```
a: A
b: B
a := b
// Polymorphic
// assignment

a.foo
// Dot call – usage
```



Scopes. Visibility control

- No public
  - All unit attributes are read-only!
  - No need for getters
- **{this}** – private
- {} – exported to none
- {A, B, C} – exported to A, B, C



```
unit Some
 {this} hidden: Data
 {} forDescendant: T
 {A, B, C} attr: T1
 foo do end
end
```

```
unit X // Zones
    {}: …
    {this}: …
    {A, B, C}: …
end
```

Unified type system. Type kinds

- Unit types
- Anchored types
    - Automatic overriding
- Generic types
    - Arrays
- Tuple types
    - Tuple expressions
- Functional (routine) types
- Multi-types (kind of unions)

```
unit Bit // Bit is a unit type
   …
end
// Anchored types: the same as
anchor1: as this
anchor2: as foo
anchor3: as attr
foo: Type do end
attr: Type
// arrays with () brackets!!!
a: Array[Type] is (Type, Type)
a(index) := Type
t is a(index)

x: (T1, T2, T3) // Tuple types
y: (f1: T1, T2, f3: T3)

func1: rtn foo // Routine types
func2: rtn (T1, T2): T3

z: T1 | T2 | T3 // Multi-type
```

Modules - singletons

- Structured approach to static
- Kinds of modules
  - 1 object per program – global module
  - 1 object per hierarchy of units
  - 1 object per routine

```
unit B
 goo do … end
end
```

```
B.goo (…)
// B is a global module

unit A use B
/* B is a module for the hierarchy
of units */
  foo do
       B.goo
  end
end

foo use B do
// B is a module for procedure foo
       B.goo
end

unit C use B as bb
  foo do
       bb.goo
  end
end
```

Inside units - definitions

Routines can be procedures or functions

- `a do end` /* that is a procedure without parameters, one may put () after routine name*/
- `foo: T do end` /* that is a function without parameters which returns an object of type T*/

Unit attributes can be variable (default) or constant

- `variable: Type`
- `const constant: Type is expression`

Routines may have locals which can be also variable or constant (default)

- `var variable is expression`
- `constant is expression`

Inside units - example

var = mut
const = let ☺

```
unit X
  const constant1: Type is someExpression
  const constant2 is someExpression
  variable0: Type
  variable1: ?Type // variable1 is explicitly non-initialized.
  variable2 is someExpression
  variable3: Type is someExpression
  routine do
       routineConstant1: Type is someExpression
       routineConstant2 is someExpression
       var routineVariable1: Type is someExpression
       var routineVariable2 is someExpression
  end
  init do
    variable0 := someExpression // That is an assignment
    // constant1 := someExpression // Compile time error
  end
end
x is X; y is X.variable0
```

How to build a program?

Entry points:

- Anonymous procedure: First statement is the entry point

```
StandardIO.put("Hello world!\n")
routine (("ha-ha-ha"))
```

- Visible stand-alone procedure

```
routine(strings: Array[String]) do
end
```

- Initialization procedure of some unit

```
unit C
    init do end
end
```

Global context:

- All top level units and stand-alone routines are mutually visible
- Name clashes are resolved outside of the language
- Visibility if units is also a feature of environment – not part of the language

----

Source 1:

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

Source 2:

```
  goo do end
```

Source 3:

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

Operators – if & loop

- One conditional statement and one loop
- 2 forms of conditional statements
- 2 forms of the loop

If-then can be dropped off

```
if condition is
  true: thenAction
  false: elseAction
end
```

```
if condition then
       thenAction
else
       elseAction
end

if a is
   T1: action1 // T1 is type
   E2: action2 // E2 is expression
   else action3
end

while index in 1..10 do
   body
end

do
   body
while condition end
```

Operators – super block

- Sequence of statements potentially decorated with pre and post conditions and errors (exception) handling
- May be nested
- Any routine is a named block with optional parameters

```
require
        predicate_1
        predicate_2
do
        statement_1
        statement_2
        when Type_1 do …
        when e: Type_2 do …
        when expr_1 do …
        else …
ensure
        predicate_3
        predicate_4
end
```

Systematic assertions and more …

```
unit Stack [G]
  push (element: G)
    ensure
       count = old count + 1
  pop: G
    require
       count > 0
    ensure
       count = old count - 1
  count: Integer
invariant
  count >= 0
end // Stack

s is Stack [Integer]
s.push (5)
var x is s.pop
x := s.pop
```

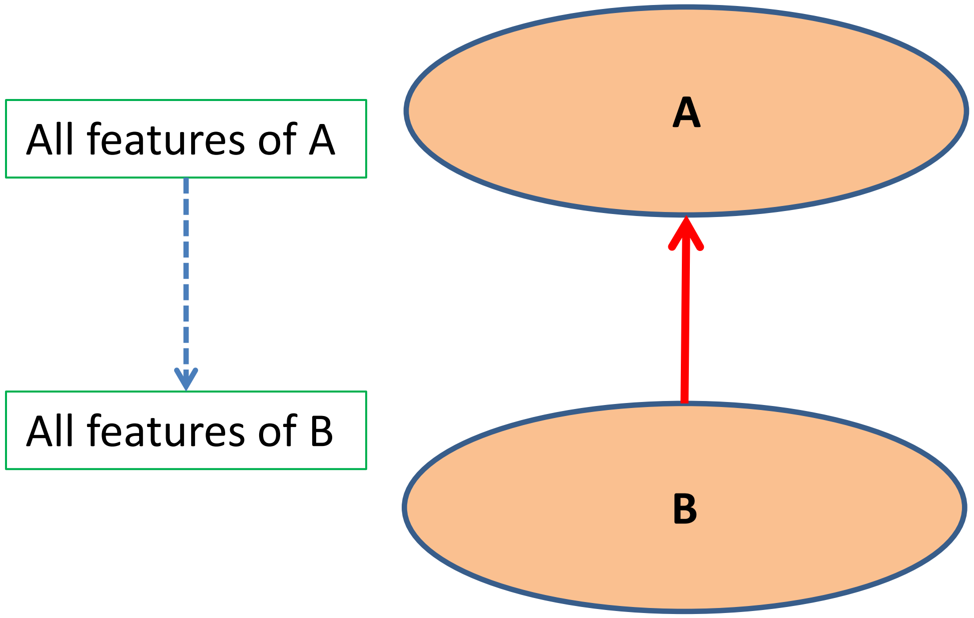
- require – routine precondition
- ensure – routine postcondition
- invariant – unit (class) invariant
- Type (unit) specification (interface) - list of all publically available members (features)
- *Duck typing – if type spec 1 conforms to the type spec 2 then duck typing can be used

1. Object created (memory allocated, initializer invoked, invariant checked)
2. Routine calls
   1. Check preconditions
   2. Execute the body
   3. Check invariant
   4. Check postconditions
3. Object disposed

Inheritance again …

<u>Unit feature (member):</u>

- Name
- Scope (visibility)
- Routine -> signature
  - Internal
    - With body
    - No body (abstract) - virtual
  - External (runtime properties)
- Attribute -> type



B conforms to A if there is a path in the inheritance graph from B to A

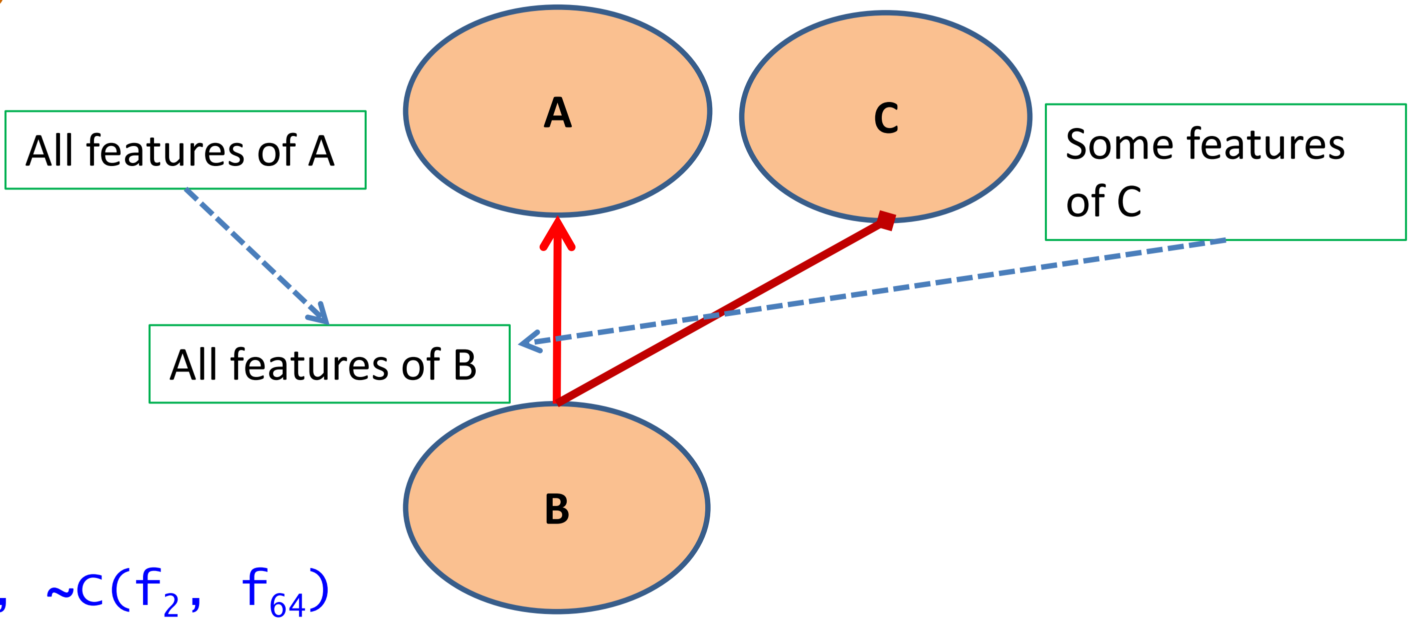
We can change (adapt) while inheriting

- Name
- Scope
- For routines:
  - Override with new signature (covariant overriding)
  - Override with new body – internal, external, virtual (no body)
- For attributes - new type (covariant overriding)

And again inheritance …

```
unit A
  foo do …  end
end
unit C
   f₁ f₂ f₃ … fₙ
end
unit B extend A, ~C(f₂, f₆₄)
end
var a is A
var c is C
var b is B

a := b  // OK!
c := b  // Compile time error!
b.f₂    // OK!
b.f₁    // Compile time error!
```



Inheritance:

- Conformant (polymorphic assignment OK)
- Non-conformant inheritance (may selectively inherit particular features)

Feature call

What is a+ b? => a.+(b)

What is ++a? => a.++()

Infix or prefix operators are just syntax sugar of the feature call (member access/invocation)

The dot call is the basic control mechanism !

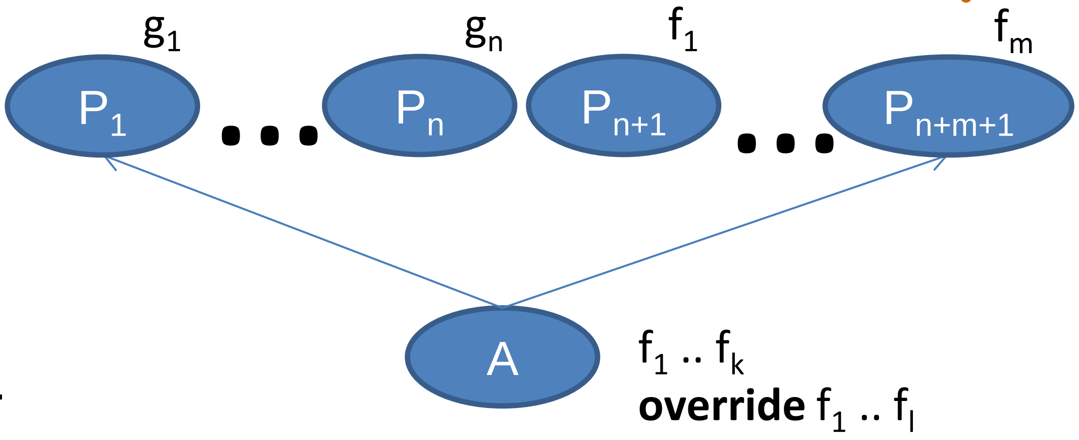
```
        target.foo (expr₁, expr₂, … exprₙ)
        foo (target, expr₁, expr₂, … exprₙ)
```

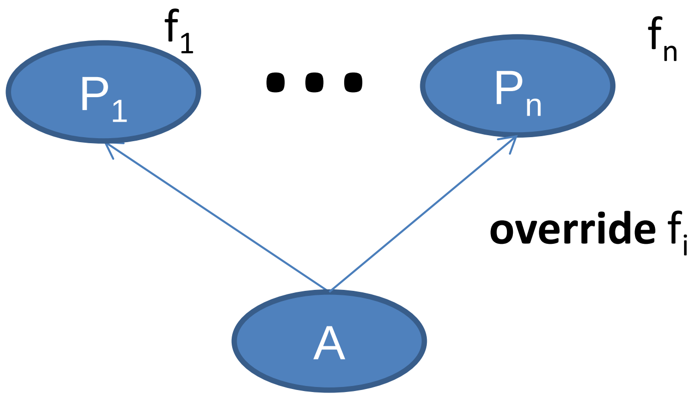
Operation signs should be used as names of routines

```
unit Comparable
   < (other: as this): Boolean virtual
   > (other: as this): Boolean => this <= other
   …
end
```

Approach to inheritance, feature call validity-1

- **Override in a unit:**
  - g_i is identical to g_j then only one g is inherited
  - g_1 .. g_n are inherited as is
  - f_1 .. f_k are introduced in A, new features
  - $_l \leq {}_m$, let f_1 .. f_l override some of f_1 .. f_m based on signature conformance then remaining (not overridden) of f_1 .. f_m are inherited as is
- **Override while inheriting**:
  - f_i will override f_1 .. f_k , where $_k < {}_n$, based on signature conformance
  - then A will have f_1 .. f_{n-k+1} features
- **Feature call validity**
  - Call is valid when it can be unambiguously resolved!
  - There is only one visible f in A with the signature (T_1..T_n) to which (ET_1..ET_n) conforms

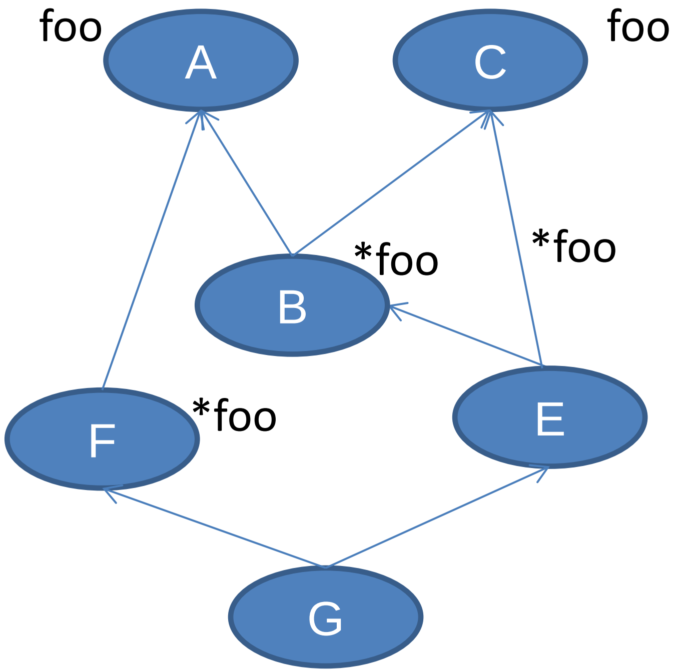




```
// P1..Pn – base units for A
// E1..En – expressions of types ETi
a is A
a.f(E1, .. En)
// Is it a valid feature call?
```

Approach to inheritance, feature call validity-2

- High-level approach: multiple inheritance with overloading and conflicting feature versions while checking feature call validity per call.
- Mandatory validity check for the inheritance graph :
  - No cycles in inheritance graph
  - All polymorphic version conflicts resolved (‘select’)



```
virtual unit A
    foo (T) virtual
end
unit C
    foo (T) do end
end
unit B extend A, C
    override foo (T) do end
end
unit E extend C, B
    override C.foo
end
unit F extend A
    override foo (T1) do end
end
unit G extend F, E
    use E.foo
end
```

Reference and value objects

- Unit is just a definition of all type members (features) It may not prescribe the form of objects
- Implicit boxing/unboxing for assignments
  - ref1 := ref2
  - val1 := val2
  - ref := val
  - val := ref

```
unit A
  …
end
var ar is ref A /* ar will be the reference
object */
var av is val A /* av will be the value
object*/
a is A // Nature of a? ☺ Default is ref!
val unit Integer … end
i is 5
unit B extend A … end
br is ref B
bv is val B

ar := av // &clone(av)
av := ar // ar^ field by field copy
ar := br // move ref
av := bv // bv field by field copy
```

Null-safety and non-initialized attributes

Key principles:

- Every entity must be initialized before any access to its attributes or routines (features/members)

```
e1 is 5 // Type of e1 is deduced from 5
e2: Type is Expression /* Type of Expression must conform to Type*/
unitAttr: Type /* init must assign value to untiAttr*/
```

- If one needs to declare an entity with no value, it is not possible to access its attributes or routines.

```
entity: ?A // entity has no value!!! Type?
```

- There must be a mechanism how to check that some entity is a valid object of some type and safe access to its attributes/routines can be granted

```
if entity is A then /* check if entity is of type A or its descendant and only then deal with it */
        entity.foo
end
```

- Entity which was declared as no-value entity may loose its value

```
? entity // detach the entity.
```

- Not able to assign

```
a: A is entity // Compile time error!
```

- Works for value type

```
i: ?Integer
i := i + 5 // Compile time error!
if i is Integer then i := i + 5 end
```

- There is no NULL/NIL/Void at all ☺

Duck typing

If an object has some particular feature or features then they can be called – the key idea.

Runtime check if the call is possible

1. To check
2. If check passes then call

In other words duck tying is a special case of dynamic type check for some anonymous unit (class)

```
foo (object: Any) do
   if object is
       unit
         attribute: T1
         function1 : T2
         function2 (p: T1): T2
         procedure1
         procedure2 (p:T1)
       end
   then
   /* type of object here is unit
described above */
       var t1 is object.a
       var t2 is object.function1
       t2 := object.function2 (t1)
       object.procedure1
       object.procedure2 (t1)
   end
end
```

Type conversions and setters

- Explicitly defined!
- From-conversion and to-conversion
  - One or both may be defined
- Aligned with assignment
- Setter is an assignment procedure for unit attribute

```
unit A
   := (b: B) do
// ‘from’ conversion procedure
   end
   := (): B do
// ‘to’ conversion function
   end
   attr: as this := (as this)
end
unit B
end
var a is A
var b is B
a := b // a.:=(b)
b := a // b := a.:=()
foo (b: B) do … end
foo (a)
a.attr := a // Setter for attr called
```

Type system foudation

```
val unit Bit
   const is // Bit is just 0 or 1
    0b0, 0b1
   end
   pure & alias and (other: Bit): as this => if this = 0b0 do 0b0 elsif
other = 0b0 do 0b0 else 0b1

   pure | alias or (other: Bit): as this => if this = 0b1 do 0b1 elsif
other = 0b1 do 0b1 else 0b0

   pure ^ alias xor (other: Bit): as this => if this = other do 0b0
else 0b1

   pure ~ alias not (): Bit => if this = 0b0 do 0b1 else 0b0
   …
end // Bit
val unit Bit [N: Integer]
   {} data: val Array [0 .. N-1, Bit] // Bit field
   ...
end
```

All unit types relies on Bit [N]

Constant objects

- Every unit may define all known constant objects using **const is**
- Integer.1 is a valid constant object of type Integer
- To skip unit name prefix use **use const**

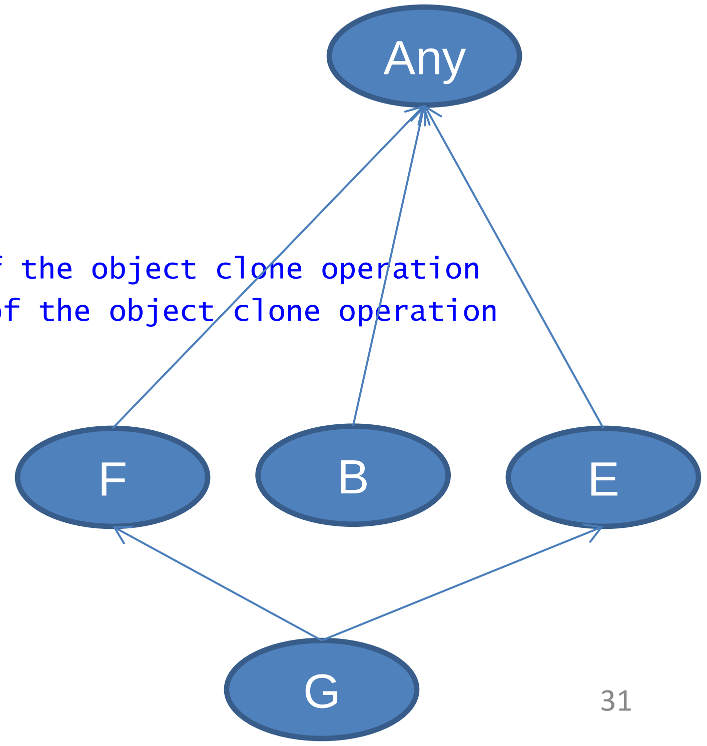
```
val unit Integer extend
    Integer [Platform.IntegerBitsCount]
…
end
val unit Integer [BitsNumber: Integer] extend
Numeric, Enumeration
   const minInteger is - (2 ^ (BitsNumber - 1))
   const maxInteger is 2 ^ (BitsNumber - 1) - 1
   const is /* That is ordered set defined as range
of all Integer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

Constant objects - examples

```
unit WeekDay
   const is Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, 
Sunday end
end
use const WeekDay foo (Monday)
foo (day: WeekDay) is
   if day is
      Monday .. Friday: StandardIO.put (“Work day – go to the 
office!\n”)
     Saturday, Sunday: StandardIO.put (“WeekEnd – do what you like!\n”)
   end
end
unit A
   const is a1.init, a2.init (T), a3.init (T1, T2)
   end
   init is end
   init (arg: T) is end
   init (arg1: T1; arg2: T2) is end
end
const x is A.a1
y is A.a2
```

Standard library basics: everything is defined

```
virtual unit Any use const Integer, Real, Boolean, Character, Bit, String
   /// Shallow equality tests
   = (that: ? as this): Boolean foreign
   final /= (that: ? as this): Boolean do return not ( this = that) end
   = (that: as this): Boolean foreign
   final /= (that: as this): Boolean do return not ( this = that) end
   /// Deep equality tests
   == (that: ? as this): Boolean foreign
   final /== (that: ? as this): Boolean do return not ( this == that) end
   == (that: as this): Boolean foreign
   final /== (that: as this): Boolean do return not ( this == that) end
   /// Assignment definition
   hidden := (that: ? as this) foreign
   hidden := (that: as this) foreign
   /// Utility
   toString: String foreign
   sizeof: Integer foreign ensure return >= 0 end
end // Any
unit System is
   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
unit Platform is
   const IntegerBitsCount is 32
   const RealBitsCount is 64
   const CharacterBitsCount is 8
   const BooleanBitsCount is 8
   const PointerBitsCount is 32
   const BitsInByteCount is 8
end // Platform
```



Standard library basics: everything is defined

```
val unit Boolean extend Enumeration is
          const is false.init (0), true.init (1) end
          override < (other: as this): Boolean => not this => other
          override = (other: as this): Boolean => this.data = other.data
          succ: as this => if this then false else true
          pred: as this => if this then false else true
          override const first is false
          override const last is true
          const count is 2
          ord: Integer => if this then 1 else 0
          override sizeof: Integer => Platform.BooleanBitsCount / Platform.BitsInByteCount
          & alias and (other: as this): Boolean =>
                    if this then if other then true else false else false
          | alias or (other: as this): Boolean =>
                    if this = false then if other then true else false else true
          ^ alias xor (other: as this): Boolean =>
                    if this then if other then false else true else if other then true else false
          => alias implies (other: as this): Boolean => not this or other
          ~ alias not : Boolean => if this then false else true
          toInteger: Integer => if this then 1 else 0
          init (value: as this) do data := value.data end
          init do data := 0xb end
          {this} init (value: Integer) require value in 0..1 do data := value end
          {this} data: Bit [Platform.BooleanBitsCount]
invariant
          this and this = this /// idempotence of 'and'
          this or this = this /// idempotence of 'or'
          this and not this = false /// complementation
          this or not this = true /// complementation
end // Boolean
```

Extended overloading

Two units are different when they have different names or they have different number of generic parameters

```
i1: Integer is 5
i2: Integer[8] is 5

s1: String[3] is
“123”
S2: String is “123”

a1: Array[Integer, 3]
is (1, 2, 3)
a2: Array [Integer]
is
(1, 2, 3)
a3: Array [Integer,
(6,8)] is (1, 2, 3)
```

```
val unit Integer extend Integer
[Platform.IntegerBitsCount] … end
val unit Integer [BitsNumber: Integer] … end
virtual unit AString   /* String abstraction */
… end

unit String [N:Integer] extend AString, Array
[Character, N] /* Fixed length string*/ … end
unit String extend Astring /* Variable length
String*/ … end

virtual unit AnArray [G] /* One dimensional
array abstraction*/ … end
unit Array [G->Any init (),
N: Integer|(Integer,Integer)]
extend AnArray [G] /* Static one dimensional
array*/ … end
unit Array [G -> Any init ()] extend AnArray
[G] /* Dynamic one dimensional array*/ … end
```

Unit extensions

- All sources are compiled separately
- Smart linking is required to support valid objects creation
- Source4 validity depends on what sources are included into the assembly

```
Source1:
unit A
   foo do end
end
Source2:
extend unit A
   goo do end
end
Source3:
extend unit A extend B
   override too do end
end
unit B
   too do end
end
Source4:
a is A
a.too
a.foo
a.goo
```

Double dispatch. Multiple overriding (I)

```
// Source #1
virtual unit Figure
     inscribedInto (other: Figure): Boolean virtual
end
// Source #2
unit Circle extend Figure
 override inscribedInto (other: Circle): Boolean do … end
end
// Source #3
unit Triangle extend Figure
 override inscribedInto (other: Triangle): Boolean do … end
end
extend unit Circle
 override inscribedInto (other: Triangle): Boolean do … end
end
// Source #4
a: Array [Figure] is (Circle, Triangle)
if a(1).inscrinedInto (a(2)) then … end
```

Double dispatch. Multiple overriding (II)

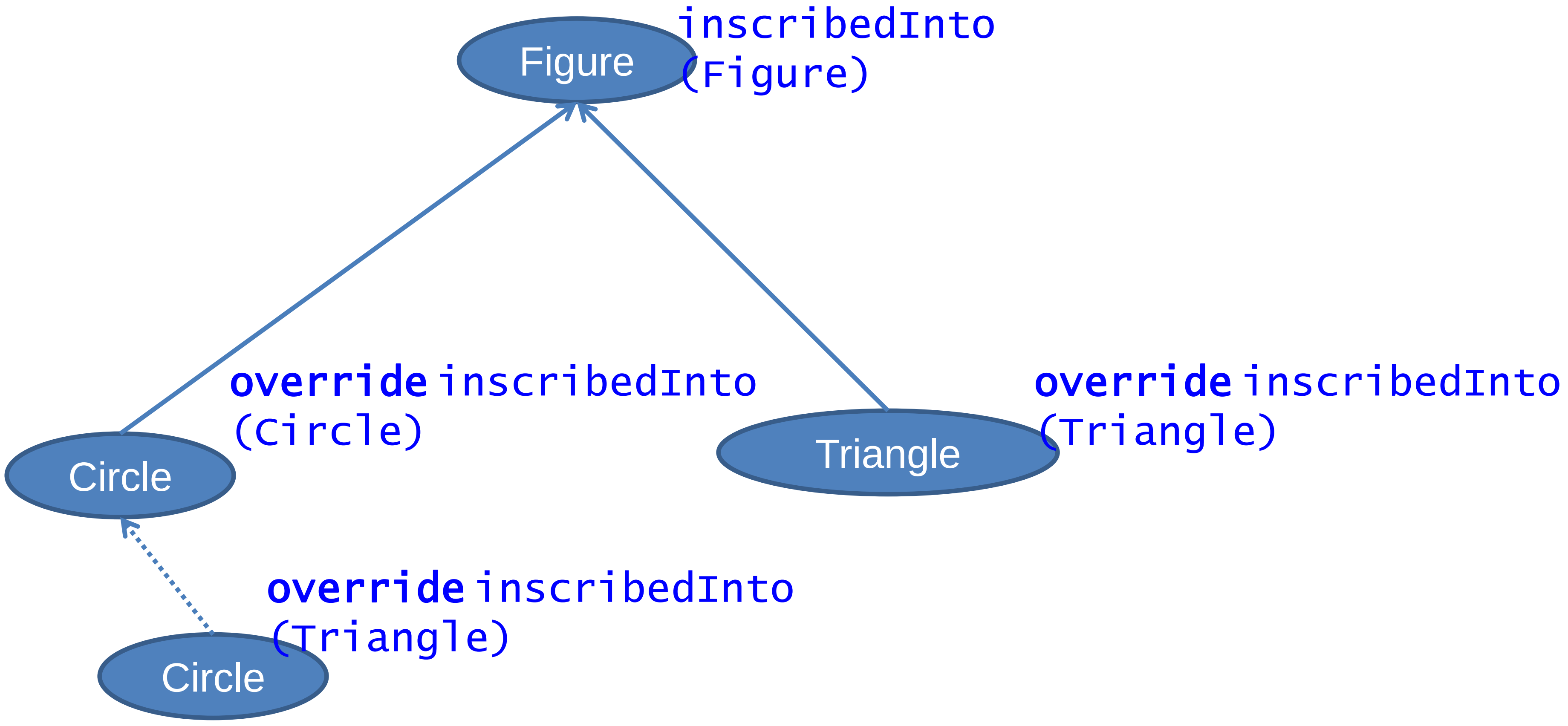


Figure
inscribedInto (Figure)
Circle
override inscribedInto (Circle)
Triangle
override inscribedInto (Triangle)
Circle
override inscribedInto (Triangle)

```
a: Array [Figure] is (Circle, Triangle)
if a(1).inscrinedInto (a(2)) then … end
```

Call to inscribedInto is valid if and only if for every dynamic type of a(1) there is version of inscribedInto with the signature to which call inscribedInto (a(2)) conforms to

Generics - example

- Standalone routines can be parameterized by type and /or value

```
x1 is factorial1 [Integer] (3) /* call to
factorial1 function will be executed at run-
time */

x2 is factorial2 [3] /*This call can be
processed at compile-time!!!*/

factorial1 [G->Numeric] (x: G): G do
   if x is
      x.zero, x.one: return x.one
   else
      return x * factorial1 (x - x.one)
   end
end

factorial2 [x:Numeric]: as x do
   if x is
      x.zero, x.one: return x.one
   else
      return x * factorial2 [x - x.one]
   end
end
```

Dining philosophers - example

```
philosophers is (concurrent Philosopher ("Aristotle"), concurrent Philosopher ("Kant"), concurrent
Philosopher ("Spinoza"), concurrent Philosopher ("Marx"), concurrent Philosopher ("Russell"))
forks is (concurrent Fork (1), concurrent Fork (2), concurrent Fork (3), concurrent Fork (4), concurrent
Fork (5))
require
   philosophers.count = forks.count or else philosophers.count = 1 and then forks.count = 2
   /* Task is valid, if # of forks is eual to the # of philosophers or if there is only 1 philosopher
then # of froks is equal to 2*/
do end
while true do /// Let them eat forever. Other algorithms may be applied
   while seat in philosophers.lower .. philosophers.upper do
      StandardIO.put ("Philosopher '" + philosophers (seat).name + "' is awake for lunch\n")
      eat (philosophers (seat), forks (seat), forks (if seat = philosophers.upper then forks.lower else
seat + 1)
   end
end
eat (philosopher: concurrent Philosopher; left, right: concurrent Fork) do
   /* Procedure with 3 concurrent parameters. Every call to eat creates a critical section which is
parameterized by required resources to enter it. When all resources are captured then the call is being
made having all resources in the exclusive access within the procedure */
   StandardIO.put ("Philosopher '" + philosopher.name + "' is eating with forks #" + left.id + " and #" +
right.id + "\n")
end
unit Philosopher
   name: String
   init (aName: as name)
end
unit Fork
   id: Integer
   init (anId: as id)
end
```

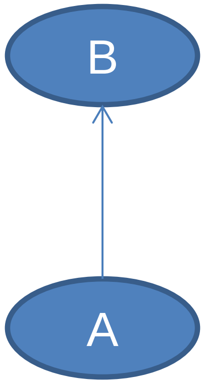
Summary

Key concepts of SLang

- Units, modules, standalone routines, usage-inheritance-typification
- Alternative approach to inheritance
- Systematic approach to assertions
- NULL-safety and non-initialized data 2 in 1
- Constant objects as the foundation for the uniform type system
- Extended overloading
- Concurrency mechanism

Conformance

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
```

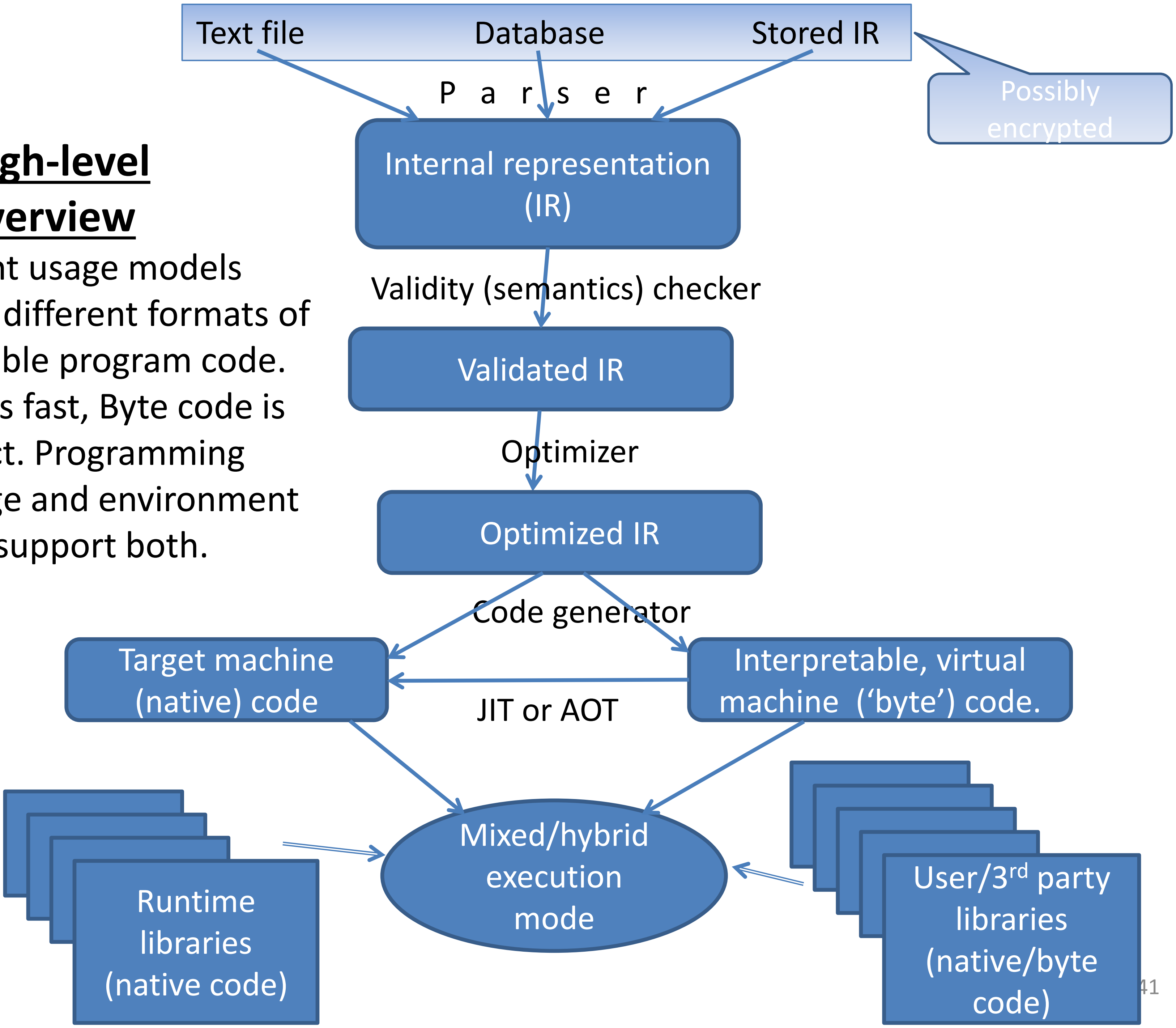
goo $(T_1, T_2, \ldots T_n)$

foo $(U_1, U_2, \ldots U_n)$

if for $_i$ in $_1 \ldots _N$
  U_i conforms to T_i

(I) High-level overview

Different usage models require different formats of executable program code. Native is fast, Byte code is compact. Programming language and environment should support both.


Text file
Database
Stored IR
Possibly encrypted
Parser
Internal representation (IR)
Validity (semantics) checker
Validated IR
Optimizer
Optimized IR
Code generator
Target machine (native) code
Interpretable, virtual machine ('byte') code.
JIT or AOT
Mixed/hybrid execution mode
Runtime libraries (native code)
User/3rd party libraries (native/byte code)

(II) Execution targets, usage models

Script

Complicated program

- Server(enterprise) => speed, concurrency, power consumption
- Desktop(single user) => speed
- Mobile => code size, power consumption
- Embedded, real-time => code size, speed, no GC delays
- Ultra mobile (IoT) => code size, power consumption

Code reuse and reliability.
Rapid application development.

JIT & AOT compilation leads to increase of power consumption on device.
Native code leads to code size growth (can be optimized with going down to 16 or 8 bit coding.
So, hybrid execution mode allows to cover all target segments.

(III) The Slang language: we all speak slang, so let's program in Slang!

Scripting – ability to create sequence of statements. Works well for mobile, WEB, IoT programming. For beginners – just write your code. But all libraries used are protected from incorrect usage with predicates.

Code reuse

- Class, module, type – 3 in 1. Unit is the approach to organization of the SW which supports separate compilation, singletons, inheritance. This works well for server, desktop and mobile segments programming
- New scheme of multiple inheritance with overloading and conflicts resolution. One concept makes programming simpler.
- Unit extensions. Programmer can add new routines and attributes into already compiled units.

Reliability

- No NULL at all. No runtime checks as every valid reference is valid.
- No non-initialized data for value and reference entities. It works well if HW support be provided – tagged architecture.
- Predicates (preconditions, postconditions, invariants). Ease of debugging. There is a limited set of runtime errors and for every error is fully know where the error occurred, why and in many cases it is straightforward how to fix it.

Concurrency

- Language level – one keyword and a special synchronization mechanism based on procedure and function calls. Dead-locks prevention mechanism.
- Auto-par – compiler level.
- 3rd party libraries like OpenMP, MPI

Ease of code development

- Functional programming in place
- Type inference

Vision

Concurrent and sequential programming with units protected with predicates is to become industry standard for the software development

Next is software verification

Next is software synthesis using neural networks techniques