Programozási nyelvek és paradigmák

Típusmegfelelés és típuskonverziók

Kozsik Tamás (2020)

Privát öröklődés

- Nem vezet be altípusosságot
- Csak a kód öröklődéséért
- Non-conforming inheritance: COMPLEX $\not<:$ MATH

```
deferred class COMPLEX
inherit {NONE}
    MATH
feature
...
invariant
    arg < 2*Pi
    arg >= 0
end
```

Kifejtett típusoknak sincsenek altípusai

- Csak saját maga és egy megfelelő kapcsolt típus
- Non-conforming inheritance: TRIPLE $\not<:$ PAIR
- Egy referencia bázistípusnak altípusa lehet: PAIR $<:$ ANY

```
expanded class PAIR
feature
    a, b: INTEGER
end
```

```
(expanded) class TRIPLE
inherit PAIR
feature
    c: INTEGER
end
```

Típuskonverziók

- Upcast
- Downcast
- Egyéb konverziók

Típuskonverziók

- Upcast
- Downcast
- Egyéb konverziók

- Explicit vagy implicit?
- Konverzió vagy csak újraértelmezés?

Elnevezések

- type conversion
- type cast
- type coercion

Különböző nyelvekben mást és mást jelenthetnek!

Upcast – altípusosság

```
class COMPLEX ...

class POLAR_COMPLEX inherit COMPLEX ...
```

- Altípusosság: POLAR_COMPLEX <: COMPLEX
- Implicit, újraértelmezés

```
local
    c: attached COMPLEX
    pc: attached POLAR_COMPLEX
do
    create {POLAR_COMPLEX} c
    create pc
    c := pc
end
```

Downcast – dinamikus típusellenőrzéssel

- Explicit, újraértelmezés

```
local
    c: attached COMPLEX
    pc: attached POLAR_COMPLEX
do
    create {POLAR_COMPLEX} c
    pc := c    -- fordítási hiba
    if attached {POLAR_COMPLEX} c as p then
        pc := p
    end
end
```

Konverzióra hasonlít: típusozott literál

```
Manifest_constant ::= [Manifest_type] Manifest_value
Manifest_type     ::= "{" Type "}"
Manifest_value    ::= Boolean_constant
                    | Character_constant
                    | Integer_constant
                    | Real_constant
                    | Manifest_string
                    | Manifest_type
```

```
{REAL_32} 3.14
```

Automatikus konverzió: konverziós eljárással

```
class DATUM
create
   make, make_masnap, from_array
convert
   from_array( {ARRAY[INTEGER]} )
feature
   from_array( arr: attached ARRAY[INTEGER] )
      require arr.count = 3
      ...
      end
...
end -- class DATUM
```

Automatikus konverzió: konverziós eljárással

```
class DATUM
create
   make, make_masnap, from_array
convert
   from_array( {ARRAY[INTEGER]} )
feature
   from_array( arr: attached ARRAY[INTEGER] )
      require arr.count = 3
      ...
      end
...
end -- class DATUM
```

```
d: DATUM
...
d := <<1848,3,15>>      -- create d.from_array(<<1848,3,15>>)
```

Automatikus konverzió: konverziós függvénnyel

```
class FRACTION
create
    set,
    from_integer
convert
    from_integer({INTEGER}),
    to_real:{REAL_64}
feature
    numerator, denominator: INTEGER
    set( n, d: INTEGER ) ...
    from_integer( i: INTEGER ) ...
    to_real: REAL_64 ...
    ...
invariant
    denominator /= 0
end -- class FRACTION
```

Jobban kifejtve

```
class FRACTION
create set, from_integer
convert from_integer({INTEGER}), to_real:{REAL_64}
feature
    numerator, denominator: INTEGER

    set( n, d: INTEGER )
        require d /= 0 do numerator:=n; denominator:=d end

    from_integer( i: INTEGER ) do set(i,1) end

    to_real: REAL_64 do Result := numerator / denominator end

    divided_by alias "/" (other: attached like Current):
        attached like Current ...


    ...
invariant
    denominator /= 0
end -- class FRACTION
```

Automatikus konverzió: használat

```
class FRACTION
create
    set, from_integer
convert
    from_integer({INTEGER}), to_real:{REAL_64}
...
end -- class FRACTION


f: attached FRACTION
r: REAL_64
...
f := 3
f := f / 4    -- 3/4
r := f / 4    -- 0.1875
```

Még implementálatlan (csak ECMA)

```
class FRACTION
create set, from_integer
convert from_integer({INTEGER}), to_real:{REAL_64}
feature
    divided_by alias "/" convert (f: attached like Current):
        attached like Current ...
    ...
invariant
    denominator /= 0
end -- class FRACTION



f: attached FRACTION
...
f := 3
f := f / 4    -- 3/4
f := 4 / f    -- 16/3
```

Polimorfizmus

- Universal
  - Parametric ($\forall$ és $\exists$; korlátozott)
  - Inclusion / subtype
- Ad-hoc
  - Overloading
  - Coercion

Luca Cardelli, Peter Wegner:
On Understanding Types, Data Abstraction, and Polymorphism.
Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

Típuskompatibilitás Eiffelben

- Konformitás (altípus)
  - Az öröklődés vezeti be
  - Kivételek: privát öröklődés, kifejtett típusok
- Konvertálhatóság

A programozó határozza meg!

Nominális és strukturális típusekvivalencia

Statikus típusrendszer esetén vizsgált kérdés

- Nominális: ha beleírjuk a programba, hogy ekvivalensek
- Strukturális: ha szerkezetileg megegyeznek

Vannak nyelvek, ahol keveredik...

Pascal tömb típusai

```
type
    TA = array [1,10] of Integer;
    TB = array [1,10] of Integer;
var
    A: TA;
    B: TB;
begin
    ...
    A := B;
```

Nominális és strukturális altípusosság

- Nominális
  - Csak olyan típusok állnak relációban, amelyeknél kértük
  - Jellemzően az öröklődés mentén (OOP)
- Strukturális
  - Típusok felépítése szerinti induktív definíció
  - Alkotóelemek altípusosságából összetett típusokra
  - Függvényeknél kontravariáns argumentum
  - Olyan típusok is relációba kerülhetnek, amelyeknek nem kellene

Vannak nyelvek, ahol keveredik... (pl. Scala, OCaml)

C++ template-függvényei

```
template <typename T> int doubleX( T t ){
        return 2 * t.x;
}

struct A { int x; };
class B { public: int x; };

int main(){
        A a; a.x = 19;
        B b; b.x = 45;
        int ax2 = doubleX(a);
        int bx2 = doubleX(b);
    ...
}
```

Egy jobb példa

```
record PolarComplex
{
    double phase, magnitude;
};

record VelocityVector
{
    double phase, magnitude;
};

record VelocityVector3d
{
    double phase, magnitude, azimuth;
};
```

Polimorf művelet

```
template <typename T>
void add( T amount, T& to ){
    double amount_re = amount.magnitude * cos(amount.phase);
    double amount_im = amount.magnitude * sin(amount.phase);
    double to_re = to.magnitude * cos(to.phase);
    double to_im = to.magnitude * sin(to.phase);
    to_re += amount_re;
    to_im += amount_im;
    to.magnitude = sqrt(to_re*to_re + to_im*to_im);
    to.phase = ... atan( to_im/to_re ) ...
}
```

Típuskonstrukciók

- Elemi típus
- Sorozat / tömb
- Rendezett n-es
- Rekord
- Függvény
- Osztály

Strukturális altípusosság szabályai értékekre

- Ha $n \geq m$ és $A <: B$, akkor $A^n <: B^m$
- Ha $n \geq m$ és $\forall i \in [1..m] : A_i <: B_i$, akkor $\times_{i=0}^{n} A_i <: \times_{i=0}^{m} B_i$
- Ha a B rekord típus minden szelektora előfordul az A rekord típusban is, és minden ilyen szelektorra az A-beli típus altípusa a B-belinek, akkor $A <: B$.
- Ha $A <: A'$ és $B' <: B$, akkor $A' \to B' \ <: \ A \to B$.

Strukturális altípusosság szabályai értékekre

- Ha $n \geq m$ és $A <: B$, akkor $A^n <: B^m$
- Ha $n \geq m$ és $\forall i \in [1..m] : A_i <: B_i$, akkor $\times_{i=0}^{n} A_i <: \times_{i=0}^{m} B_i$
- Ha a B rekord típus minden szelektora előfordul az A rekord típusban is, és minden ilyen szelektorra az A-beli típus altípusa a B-belinek, akkor $A <: B$.
- Ha $A <: A'$ és $B' <: B$, akkor $A' \to B' \ <: \ A \to B$.

Észrevételek rekord típusokra:

- Egy rekord típusban a mezők megadásának sorrendje érdektelen.
- Ha A és B rekord típusokra $A <: B$, akkor az A-hoz további mezőket hozzávéve az altípusosság megmarad.

Currying

Modellezzük az $(A \times B) \to C$ függvénytípust az $A \to (B \to C)$ függvénytípussal.

A zárójelek konvenzionális elhagyásával: $A \times B \to C$, illetve $A \to B \to C$.

Currying

Modellezzük az $(A \times B) \to C$ függvénytípust az $A \to (B \to C)$ függvénytípussal.

A zárójelek konvenzionális elhagyásával: $A \times B \to C$, illetve $A \to B \to C$.

Mikor lesz $A' \to B' \to C' \ <: \ A \to B \to C$?

Currying

Modellezzük az $(A \times B) \to C$ függvénytípust az $A \to (B \to C)$ függvénytípussal.

A zárójelek konvenzionális elhagyásával: $A \times B \to C$, illetve $A \to B \to C$.

Mikor lesz $A' \to B' \to C' \ <: \ A \to B \to C$?

Mikor lesz $(A' \to B') \to C' \ <: \ (A \to B) \to C$?

Változók modellezése

$v : T$ változó modellezhető $T \times (T \to \mathrm{Unit})$ párként.

```
v: attached T
```

```
get_v: attached T assign set_v
set_v( value: attached like get_v )
```

Változók modellezése

$v : T$ változó modellezhető $T \times (T \to \mathrm{Unit})$ párként.

```
v: attached T
```

```
get_v: attached T assign set_v
set_v( value: attached like get_v )
```

Imperatív / módosítható (mutable) kontextusban ez alapján kell altípusszabályokat hozni.

$A \times (A \to \mathrm{Unit}) \quad <: \quad B \times (B \to \mathrm{Unit})$ feltétele:
$A <: B \wedge B <: A$ (azaz $A \equiv B$)

Módosítható rekord

```
struct point { int x; int y; };
```

$$\{\text{getx} : \textbf{int},\ \text{setx} : \textbf{int} \to \text{Unit},\ \text{gety} : \textbf{int},\ \text{sety} : \textbf{int} \to \text{Unit}\}$$

Strukturális altípusosság szabályai módosítható adatokra

```
struct pairA{ A u, v; };  <:  struct pairB{ B u, v; };
```

$$\{\text{getu} : A,\ \text{setu} : A \to \text{Unit},\ \text{getv} : A,\ \text{setv} : A \to \text{Unit}\}$$

$$<:$$

$$\{\text{getu} : B,\ \text{setu} : B \to \text{Unit},\ \text{getv} : B,\ \text{setv} : B \to \text{Unit}\}$$

Strukturális altípusosság szabályai módosítható adatokra

```
struct pairA{ A u, v; };  <:  struct pairB{ B u, v; };
```

$$\{\text{getu} : A,\ \text{setu} : A \to \text{Unit},\ \text{getv} : A,\ \text{setv} : A \to \text{Unit}\}$$

$$<:$$

$$\{\text{getu} : B,\ \text{setu} : B \to \text{Unit},\ \text{getv} : B,\ \text{setv} : B \to \text{Unit}\}$$

Invariancia! $A \equiv B$

Strukturális típusok Scalában (1)

```
scala> var v = new AnyRef { var n: Int = 1 }
v: AnyRef{def n: Int; def n_=(x$1: Int): Unit}
                                        = $anon$1@1b2df3aa
```

Strukturális típusok Scalában (1)

```
scala> var v = new AnyRef { var n: Int = 1 }
v: AnyRef{def n: Int; def n_=(x$1: Int): Unit}
                                        = $anon$1@1b2df3aa

scala> def to5( z: { def n_=(t: Int): Unit } ){ z.n_=(5) }
to5: (z: AnyRef{def n_=(t: Int): Unit})Unit
```

Strukturális típusok Scalában (1)

```
scala> var v = new AnyRef { var n: Int = 1 }
v: AnyRef{def n: Int; def n_=(x$1: Int): Unit}
                                        = $anon$1@1b2df3aa

scala> def to5( z: { def n_=(t: Int): Unit } ){ z.n_=(5) }
to5: (z: AnyRef{def n_=(t: Int): Unit})Unit

scala> v.n
res0: Int = 1
```

Strukturális típusok Scalában (1)

```
scala> var v = new AnyRef { var n: Int = 1 }
v: AnyRef{def n: Int; def n_=(x$1: Int): Unit}
                                        = $anon$1@1b2df3aa

scala> def to5( z: { def n_=(t: Int): Unit } ){ z.n_=(5) }
to5: (z: AnyRef{def n_=(t: Int): Unit})Unit

scala> v.n
res0: Int = 1

scala> to5(v)

scala> v.n
res2: Int = 5
```

Strukturális típusok Scalában (2)

```
scala> val u = new AnyRef { val n: Int = 1 }
u: AnyRef{val n: Int} = $anon$1@34775090
```

Strukturális típusok Scalában (2)

```
scala> val u = new AnyRef { val n: Int = 1 }
u: AnyRef{val n: Int} = $anon$1@34775090

scala> def get( p: AnyRef{ def n: Int } ) = p.n
get: (p: AnyRef{def n: Int})Int
```

Strukturális típusok Scalában (2)

```
scala> val u = new AnyRef { val n: Int = 1 }
u: AnyRef{val n: Int} = $anon$1@34775090

scala> def get( p: AnyRef{ def n: Int } ) = p.n
get: (p: AnyRef{def n: Int})Int

scala> get(u)
res3: Int = 1

scala> get(v)
res4: Int = 5
```

Strukturális típusok Scalában (2)

```
scala> val u = new AnyRef { val n: Int = 1 }
u: AnyRef{val n: Int} = $anon$1@34775090

scala> def get( p: AnyRef{ def n: Int } ) = p.n
get: (p: AnyRef{def n: Int})Int

scala> get(u)
res3: Int = 1

scala> get(v)
res4: Int = 5

scala> class X { def n = 42 }
defined class X

scala> get( new X )
res5: Int = 42
```

Strukturális típusok Scalában (3)

```
scala> class Food; class Milk extends Food
defined class Food
defined class Milk

scala> def diet( animal: AnyRef{ def prefers: Food } ) =
     |         animal.prefers
diet: (animal: AnyRef{def prefers: Food})Food

scala> val cat = new AnyRef { def prefers: Milk = new Milk }
cat: AnyRef{def prefers: Milk} = $anon$1@42ff6af7

scala> diet(cat)
res6: Food = Milk@102ee705
```

Ismétlés: kovariáns visszatérési érték Eiffelben

```
deferred class FOOD end
class MILK inherit FOOD ...

class ANIMAL
feature
    prefers: detachable FOOD do end
...

class CAT
inherit ANIMAL redefine prefers end
feature
    prefers: attached MILK do create Result end
...
```

Korrekt altípusosság

```
class SKIER
feature
    roommate: detachable like Current
end


class GIRL
inherit SKIER
end
```

LSP megsértése

```
class SKIER
feature
    roommate: detachable like Current assign set
    set( mate: detachable like Current )
        do
            roommate := mate
        end
end


class GIRL
inherit SKIER
end
```

Módosíthatatlan `roommate` referencia: LSP OK

```
class SKIER
create
    set
feature
    roommate: detachable like Current

feature {NONE}
    set( mate: detachable like roommate )
        do
            roommate := mate
        end
end


class GIRL inherit SKIER create set end
```

Duck-typing

- Dinamikus típusozású, interpretált nyelvekben, pl. Python
- Hasonlít a strukturális altípusosságra

```
def f(p):
    if p.x > 0:
        p.g(p.x)
    else:
        p.h()
```

Prototípus alapú nyelvek

- Self, *JavaScript* ($<$ ES6), Lua, Cecil stb.
- Objektumok klónozásával hozunk létre új objektumokat (nincs osztály)
- Az objektumok dinamikusan bővíthetők (nem kell öröklődés)
- Jellemzően interpretált nyelvek

```
var foo = { one: 1,
            two: 2,
            f: function(x){ return x+this.one}
          }
var bar = Object.create(foo)           // inherit from foo
bar.one = "one"                        // redefine a member
bar.three = bar.f(bar.one) + bar.two   // extend with a member
```