



Programozási nyelvek és módszerek

PROGRAMOK HELYESSÉGE

Programok helyessége

Feladat

- Algoritmust adok a megoldására
- Megoldja?
 - Teszteljük
 - Fekete doboz
 - Fehér doboz
 - Nem biztos a megoldás

Formális matematikai megközelítés szükséges!

Sir Charles Antony Richard Hoare

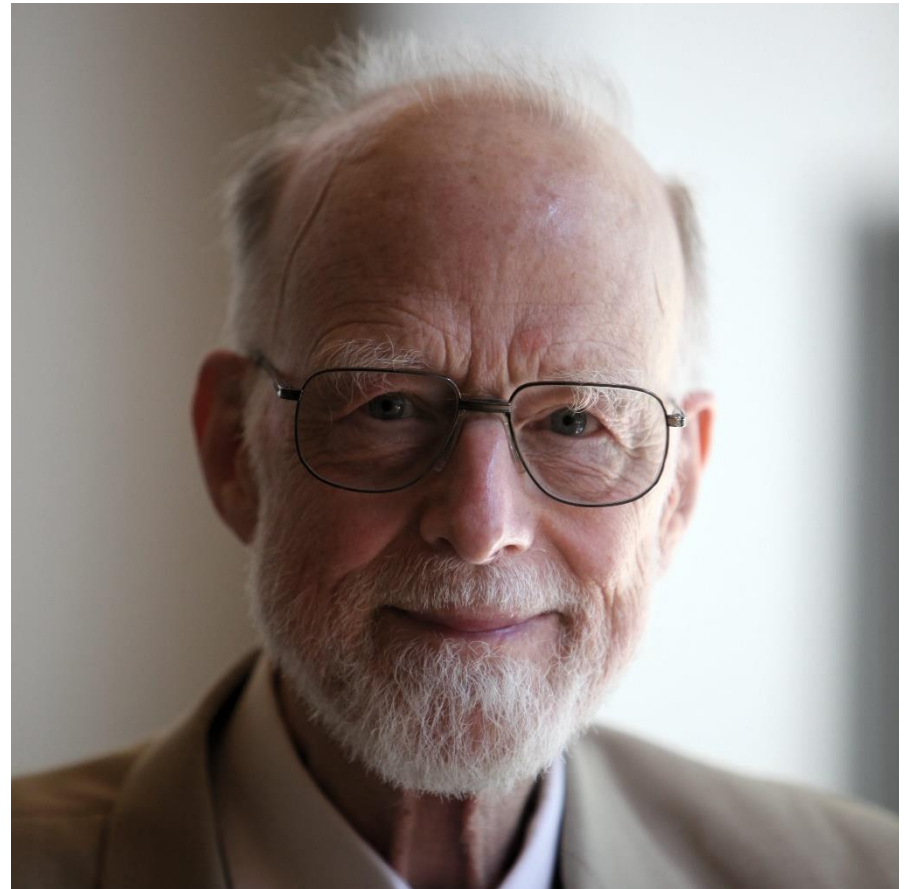
Brit tudós

Fontosabb eredmények

- Quicksort algoritmus
- „Hoare logic”
- A programhelyesség igazolására
- Strukturált programozás
- Párhuzamos programozás
- Null referencia
 - Ezért 2009-ben bocsánatot is kért

Hol

- Oxford
- Microsoft Research Lab.
(Cambridge)



Feladat specifikációja

Szerződés a felhasználó és az implementáló között

- **előfeltétel**: egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- **utófeltétel**: egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével:

- ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.

(Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)

Feladat specifikációja

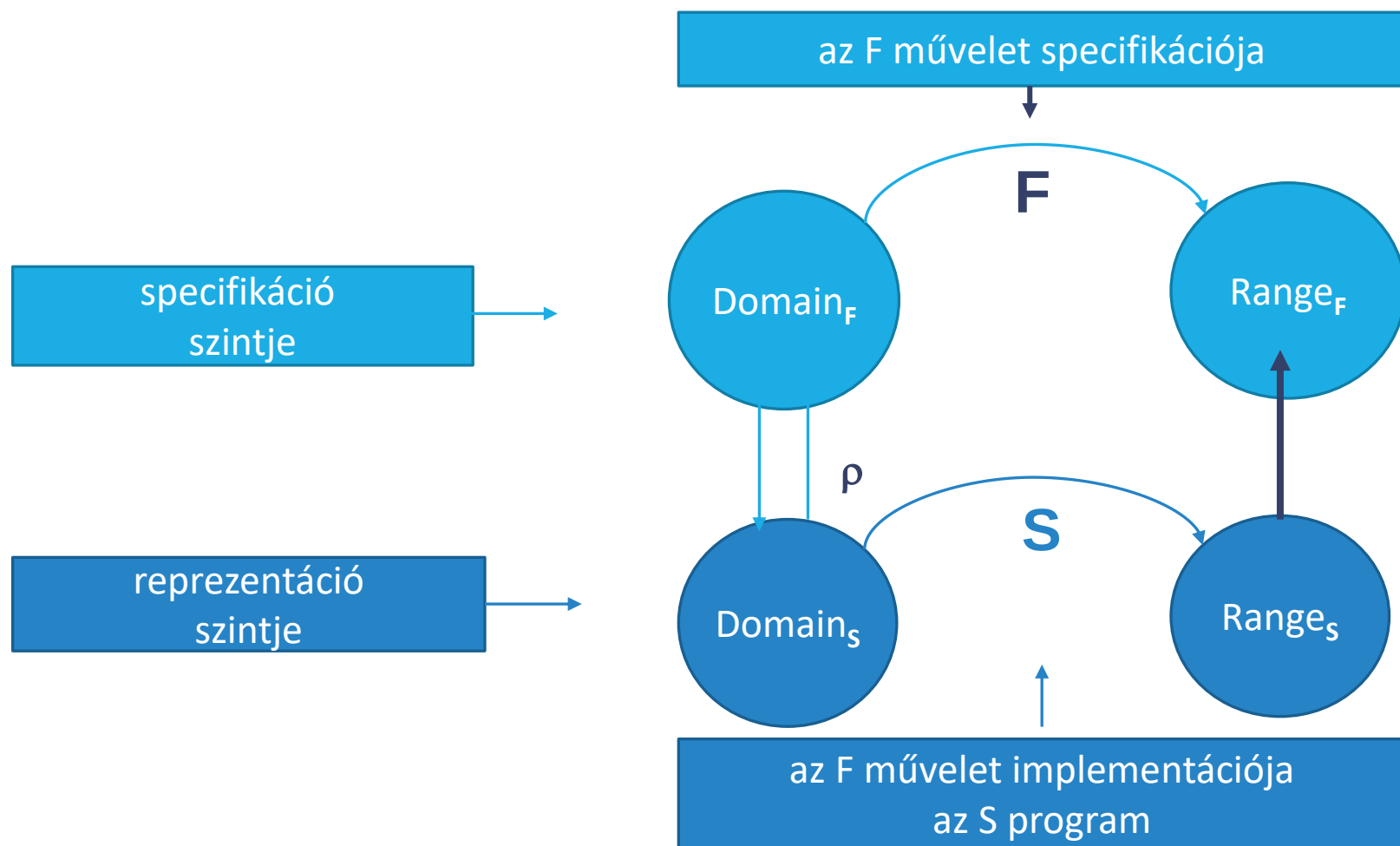
A strukturált és az objektumorientált programozási nyelvek központi fogalmává vált az absztrakció és az absztrakt adattípus.

- Ez a megközelítés azt jelenti, hogy a típust a típusspecifikációval, a típusimplementációval és a közöttük levő kapcsolatot megadó reprezentációs függvénnel definiáljuk.
- A helyességbizonyításhoz tehát olyan specifikációs módszerekre van szükségünk, amelyekkel a típusspecifikációt és a típusimplementációt adhatjuk meg.
 - Ezt nevezzük kettős specifikációnak.
 - Ebben az esetben a helyességbizonyítás során azt kell ellenőrizni, hogy az implementáció megfelel-e a specifikációnak.

A Hoare-féle specifikációban:

- A típusműveletekhez elő- és utófeltételeket rendelünk.
- A típusérték-halmazok leírása invariánsokat is tartalmazhat

A típus specifikáció és a típus kapcsolata



Lehetőségek

Az elő- és utófeltételes specifikációt alkalmazza számos programozási nyelv

- Közvetlenül pl.
 - az Alphard,
 - az Eiffel,
 - a D,
 - az Oxygene (korábban Chrome) (Object Pascal alapú)
 - a Cobra
 - ...
- Közvetve (valamilyen eszköz segítségével)
 - C#, VB a Code Contracts használatával (ez egy Microsoft Research project, a .Net Framework -be integrálva),
 - Java számos eszközzel, pl. iContract2, Contract4J, jContractor, Jcontract, Java Modeling Language (JML),
 - ...

Hoare - módszer

A feladat:

- adjuk össze egy valós elemeket tartalmazó vektor elemeit!

Kiinduló adatok:

- egy n hosszú, valós elemeket tartalmazó v vektor

Eredmény:

- s valós érték, ami tartalmazza a vektor elemeinek összegét!

Írjunk rá megoldó algoritmust!

Hoare - módszer

```
float sum(float *v, int n) {  
    float s = 0.0;  
    int i = 0;  
    while (i < n) {  
        s = s + v[i];  
        i = i + 1;  
    }  
    return s;  
}
```

Hogyan indokoljuk, hogy ez a program helyes (megoldja a feladatot)?

Feladat specifikációja

Szerződés a felhasználó és az implementáló között

- **előfeltétel**: egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- **utófeltétel**: egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével:

- ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.

(Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)

Feladat specifikációja

Állítás: egy logikai függvény a program állapotával kapcsolatosan

Például:

- $x=3$
- $y > x$
- $(x \neq 0) \Rightarrow (y+z = w)$
- $s = \sum_{(i \in 1..n)} v[i]$
- $\forall i \in 1..n : v[i] > v[i-1]$
- true

Feladat specifikációja

Állapottér

- Jelölje $\mathcal{V}$ az n hosszú, valós elemekből álló vektorok típusát! (0-tól $n-1$ -ig indexelve)
- $\mathcal{V} \times \mathcal{R}$
- Változók: $v \quad s$
- Előfeltétel: $v = v'$ és $n > 0$
- Utófeltétel: $s = (\Sigma i \mid 0 \leq i < n : v[i])$

Hoare – hármas

A programhelyesség formális indoklásához, elő- és utófeltételek használatával

Szintaxis: $\{Q\} S \{R\}$

- Q és R állítások
- S program

Jelentése: Ha kiinduláskor igaz a Q és végrehajtjuk S-t, akkor S egy olyan állapotban terminál, ahol R igaz.

Példák Hoare – hármaskokra

$\{ \text{true} \} x := 5 \{ x = 5 \}$

$\{ x = y \} x := x + 3 \{ x = y + 3 \}$

$\{ x > 1 \} x := x * 2 \{ x > 2 \}$

$\{ x = a \} \text{if } (x < 0) \text{ then } x := -x \{ x = |a| \}$

$\{ \text{false} \} x := 3 \{ x = 8 \}$

Legerősebb utófeltétel

Néhány érvényes Hoare-hármas:

- $\{x = 5\} x := x * 2 \{ \text{true} \}$
- $\{x = 5\} x := x * 2 \{ x > 0 \}$
- $\{x = 5\} x := x * 2 \{ x = 10 \mid \mid x = 5 \}$
- $\{x = 5\} x := x * 2 \{ x = 10 \}$

Mind igaz, de az utolsó a leghasznosabb:

- $x=10$ a legerősebb utófeltétel

Legerősebb utófeltétel

Definíció: Ha $\{Q\} S \{R\}$ és minden olyan R' -re, amire $\{Q\} S \{R'\}$, $R \Rightarrow R'$, akkor R az S Q -ra vonatkozó legerősebb utófeltétele

- ellenőrizzük: $x = 10 \Rightarrow \text{true}$
- ellenőrizzük : $x = 10 \Rightarrow x > 0$
- ellenőrizzük : $x = 10 \Rightarrow x = 10 \parallel x = 5$
- ellenőrizzük : $x = 10 \Rightarrow x = 10$

Leggyengébb előfeltétel

Néhány érvényes Hoare-hármas:

- $\{x = 5 \ \&\& \ y = 10\} \ z := x / y \ \{z < 1\}$
- $\{x < y \ \&\& \ y > 0\} \ z := x / y \ \{z < 1\}$
- $\{y \neq 0 \ \&\& \ x / y < 1\} \ z := x / y \ \{z < 1\}$

Mind igaz, de az utolsó a leghasznosabb, mert ez az, ami a legáltalánosabb feltételekkel engedi hívni a programot:

- $y \neq 0 \ \&\& \ x / y < 1$ a leggyengébb előfeltétel

Definíció: Ha $\{Q\} S \{R\}$ és $\forall Q' \text{ --re, ahol } Q' \Rightarrow Q$ igaz, hogy $\{Q'\} S \{R\}$, akkor Q az S R -re vonatkozó leggyengébb előfeltétele (weakest precondition).

- Jelölje ezt $wp(S, R)$.

A wp általános tulajdonságai

1. A csoda kizárásának törvénye
$$\text{wp}(S, \text{false}) = \text{false}$$
2. Monotonitási tulajdonság
ha $P \Rightarrow R$ akkor $\text{wp}(S, P) \Rightarrow \text{wp}(S, R)$
3. $\text{wp}(S, P \text{ and } R) = \text{wp}(S, P) \text{ and } \text{wp}(S, R)$
4. $\text{wp}(S, P) \text{ or } \text{wp}(S, R) \Rightarrow \text{wp}(S, P \text{ or } R)$

(Bizonyítás az érdeklődőknek az irodalomban.)

Levezetési szabályok

Értékadás

- $\{ Q \} x := 3 \{ x+y > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?
- Mi az a legáltalánosabb értéke az y-nak, amire $3 + y > 0$?
- $y > -3$

Levezetési szabályok

Értékadás

- $\{ Q \} x := 3*y + z \{ x * y - z > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?
- Hogy csináltuk az előbb?

Levezetési szabályok

Értékadás

- $\{ Q \} x := 3 \{ x + y > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Értékadási szabály:

$$\text{wp}(x := E, V) = V^{x \leftarrow E}$$

$$\{ V^{x \leftarrow E} \} x := E \{ V \}$$

$$\begin{aligned} & (x + y > 0) \quad x \leftarrow 3 \\ &= (3) + y > 0 \\ &= y > -3 \end{aligned}$$

Levezetési szabályok

Értékadás

- $\{ Q \} x := 3*y + z \{ x * y - z > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Értékadási szabály:

$$\text{wp}(x := E, V) = V^{x \leftarrow E}$$

$$\{ V^{x \leftarrow E} \} x := E \{ V \}$$

$$\begin{aligned} & (x * y - z > 0) x \leftarrow 3*y + z \\ &= (3*y + z) * y - z > 0 \\ &= 3*y^2 + z*y - z > 0 \end{aligned}$$

Levezetési szabályok

Szekvencia

- $\{ Q \} x := x + 1; y := x + y \{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?

Levezetési szabályok

Szekvencia

- $\{ Q \} x := x + 1; y := x + y \{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Szekvencia szabálya:

$$\text{wp}(S;T, R) = \text{wp}(S, \text{wp}(T, R))$$

$$\begin{aligned}\text{wp}(x:=x+1; y:=x+y, y>5) \\ &= \text{wp}(x:=x+1, \text{wp}(y:=x+y, y>5)) \\ &= \text{wp}(x:=x+1, x+y>5) \\ &= x+1+y>5 \\ &= x+y>4\end{aligned}$$

Levezetési szabályok

Elágazás

- $\{ Q \} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Intuitív megközelítés

ha a feltétel igaz: $\{ Q1 \} y := x \{ y > 5 \}$

$$Q1 = x > 5$$

else ág: $\{ Q2 \} y := -x \{ y > 5 \}$

$$Q2 = -x > 5$$

$$Q2 = x < -5$$

$$Q = x > 5 \parallel x < -5$$

Levezetési szabályok

Elágazás

- $\{ Q \}$ if $x > 0$ then $y := x$ else $y := -x$ $\{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Elágazás szabálya:

$$\begin{aligned} & \text{wp}(\text{if } B \text{ then } S \text{ else } T, R) \\ &= B \Rightarrow \text{wp}(S, R) \ \&\& \ \neg B \Rightarrow \text{wp}(T, R) \end{aligned}$$

$$\begin{aligned} & \text{wp}(\text{if } x > 0 \text{ then } y := x \text{ else } y := -x, y > 5) \\ &= x > 0 \Rightarrow \text{wp}(y := x, y > 5) \ \&\& \ x \leq 0 \Rightarrow \text{wp}(y := -x, y > 5) \\ &= x > 0 \Rightarrow x > 5 \ \&\& \ x \leq 0 \Rightarrow -x > 5 \\ &= x > 0 \Rightarrow x > 5 \ \&\& \ x \leq 0 \Rightarrow x < -5 \\ &= x > 5 \ || \ x < -5 \end{aligned}$$

Levezetési szabályok

Ciklusok

- $\{ Q \}$ while (felt) törzs $\{ R \}$
- Mi a Q leggyengébb előfeltétel?

Ciklushelyesség bizonyítása

Tekintsük először a parciális helyességet

- Nem biztos, hogy terminál, de ha igen, az utófeltétel igaz lesz
 - $\{Q\} \text{ while } B \text{ do } S \{R\}$

Keressünk egy Inv invariánst, amire

- $Q \Rightarrow Inv$
 - az invariáns kezdetben igaz
- $\{ Inv \ \&\& \ B \} S \{ Inv \}$
 - a ciklusmag minden végrehajtása megőrzi az invariánst
- $(Inv \ \&\& \ \neg B) \Rightarrow R$
 - az invariánsból és a ciklus kilépési feltételéből következik az utófeltétel

Ciklus példa

Bizonyítsuk, hogy a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

while $(j < n)$ do

$s := s + v[j];$

$j := j + 1;$

end

$\{ s = (\sum i \mid 0 \leq i < n : v[i]) \}$

Ciklus példa

Bizonyítsuk, hogy a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

$\{ \text{Inv} \}$

while $(j < n)$ do

$\{ \text{Inv} \ \&\& \ j < n \}$

$s := s + v[j];$

$j := j + 1;$

$\{ \text{Inv} \}$

end

$\{ s = (\sum i \mid 0 \leq i < n : v[i]) \}$

Ciklus invariáns keresése

Általában az utófeltétel valamilyen gyengítése

- $s = (\sum i \mid 0 \leq i < n : v[i])$

Függ valamilyen módon a ciklusváltozótól

- Tudjuk, hogy j kezdetben 0, és inkrementáljuk, amíg el nem éri n -t
- Így $0 \leq j \leq n$ valószínűleg az invariáns része

Ciklus kilépési feltétel && invariáns $\Rightarrow$ utófeltétel

- Ciklus kilépési feltétel : $j = n$

Jó ötlet: helyettesítsük az utófeltételben n -t j -vel:

- $(\sum i \mid 0 \leq i < j : v[i])$

Invariáns

- $0 \leq j \leq n \ \&\& \ s = (\sum i \mid 0 \leq i < j : v[i])$

Ciklus példa

Bizonyítsuk, hogy a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

$\{ 0 \leq j \leq n \ \&\& \ s = (\sum i \mid 0 \leq i < j : v[i]) \}$

while $(j < n)$ do

$\{ 0 \leq j \leq n \ \&\& \ s = (\sum i \mid 0 \leq i < j : a[i]) \ \&\& \ j < n \}$

$s := s + v[j];$

$j := j + 1;$

$\{ 0 \leq j \leq n \ \&\& \ s = (\sum i \mid 0 \leq i < j : v[i]) \}$

end

$\{ s = (\sum i \mid 0 \leq i < n : v[i]) \}$

Mit kell belátni?

1. $Q \Rightarrow \text{Inv}$

- az invariáns kezdetben igaz

2. $\{ \text{Inv} \ \&\& \ B \} \ S \ \{ \text{Inv} \}$

- a ciklusmag minden végrehajtása megőrzi az invariánst

3. $(\text{Inv} \ \&\& \ \neg B) \Rightarrow R$

- az invariánsból és a ciklus kilépési feltételéből következik az utófeltétel

Alphard

Az Alphardot a '70-es évek második felében fejlesztették ki.

- A cél a Hoare-féle helyességbizonyításhoz egy megfelelő specifikációs eszköz kidolgozása volt, ezért sokan a tökéletes megoldást látták benne.
- Végül azonban az implementálásig soha nem jutott el.

Mégis számos helyen oktatják a típusspecifikációs módszerek között, mert a specifikációs lehetőségei és a helyességbizonyításra való alkalmassága a legtöbb módszerhez képest sokkal kifinomultabbak.

Alphard

Az Alphard a típusokat formoknak nevezi. Egy típust a következő módon kell megadni:

- form típusnév (formális paraméterek) =
beginform
specifications
...
representation
...
implementation
...
endform;

Alphard

```
form istack (n: integer) =  
beginform  
  specifications  
    requires n > 0;  
    let istack=<...,xi,...> where xi is integer;  
      -- egészekből álló sorozat  
    invariant 0 <= length (istack) <= n;  
    initially istack = nullseq;  
      -- az absztrakt objektum kezdeti tulajdonságai
```

Alphard

```
function push (s: istack, x: integer)
    pre 0 <= length (s) < n
    post s = s' ~ x,  -- konkatenáció
pop (s: istack)
    pre 0 < length (s) <= n
    post s = leader (s'),
top (s: istack) returns x: integer
    pre 0 < length (s) <= n
    post x = last (s),
isempty (s: istack) returns b: boolean
    post b = (s = nullseq);
```

representation

...

implementation

...

endform;

Alphard

A `requires` a formális paraméterekre vonatkozó megszorításokat tartalmazza.

A `let` kulcsszó után a típus leírására használt absztrakt adattípust adjuk meg (jelen esetben az integereket tartalmazó sorozatot).

Az `invariant` a specifikáció szerinti típusinvariánst írja le.

- A példányosítás majd a konkrét térben fog megtörténni.

A reprezentációs függvény, amelyet a `representation`-ben kell megadni, fog visszaképezni erre az absztrakt adattípusra.

Alphard

Minden konkrét térben inicializált objektumra alkalmazzuk a reprezentációs függvényt és az így kapott „absztrakt objektumoknak” ki kell elégíteniük az `initially`-ban megadott feltételt.

Az `initially` az absztrakt kezdeti objektum tulajdonságait írja le.

A `function` záradékban az absztrakt adattípus műveleteit specifikáljuk elő-utófeltételekkel.

Alphard

```
form istack (n: integer) =  
  beginform  
    specifications  
    ...  
  representation  
    unique  v: vector (integer, 1, n)  
             sp: integer init sp <- 0;  
    rep (v, sp) = seq (v, 1, sp);  
    invariant 0 <= sp <= n;  
  states  
    empty when sp = 0,  
    normal when 0 < sp < n,  
    full when sp = n,  
    error otherwise;
```

Alphard

Implementation

```
body push out (s.sp=s.sp'+1 and s.v=alpha(s.v',s.sp,x)) =  
    empty, normal:: (s.sp<-s.sp+1; s.v[s.sp] <- x);  
    otherwise:: FAIL;  
body pop out (s.sp = s.sp' - 1) =  
    normal, full:: s.sp <- s.sp - 1;  
    otherwise:: FAIL;  
body top out (x = s.v [s.sp]) =  
    normal, full:: x <- s.v [s.sp];  
    otherwise:: FAIL;  
body isempty out (b = (sp = 0)) =  
    normal, full:: b <- false;  
    empty:: b <- true;  
    otherwise:: FAIL;  
endform;
```

Alphard

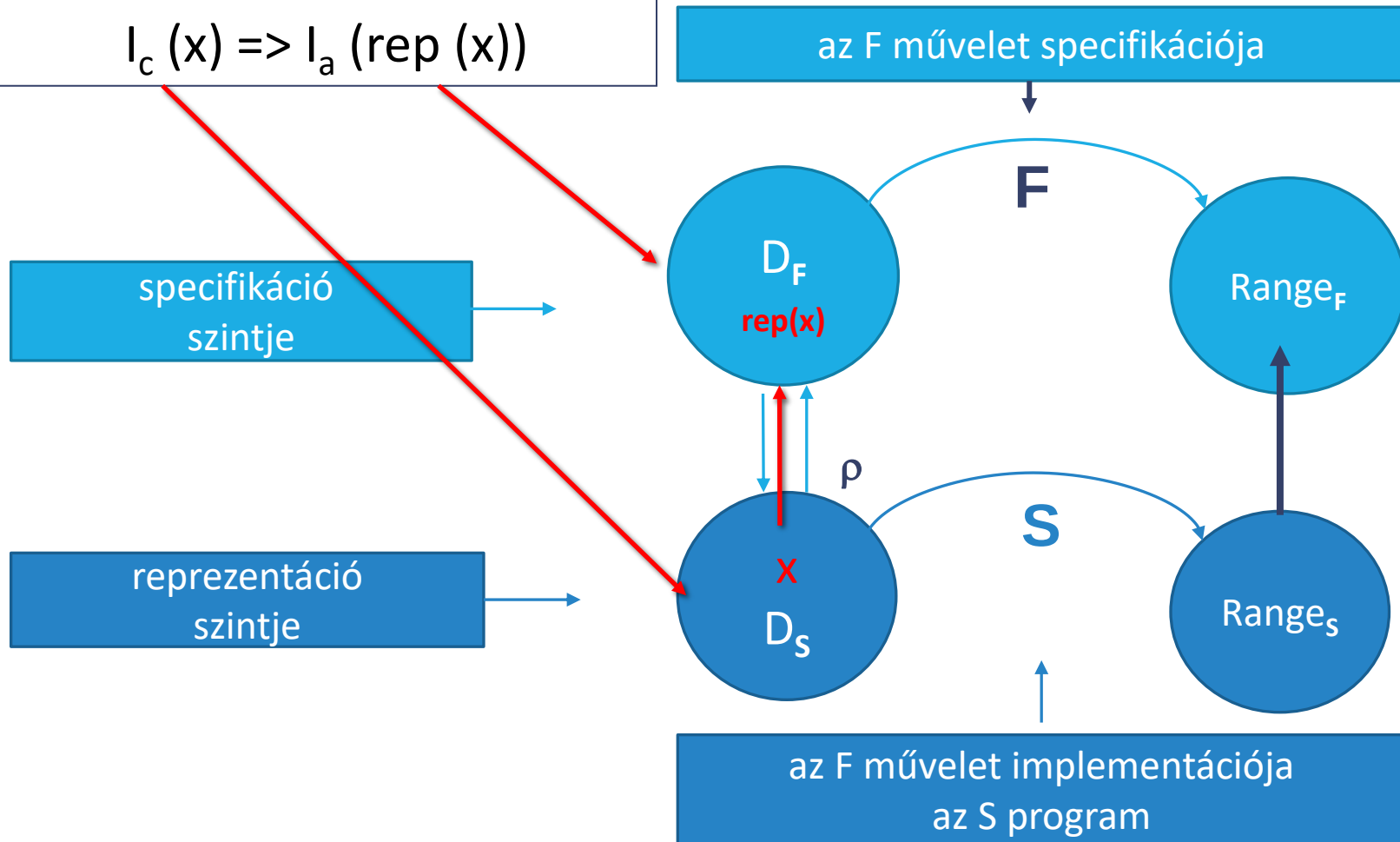
Hogyan láthatjuk be az így specifikált adattípus (form) helyességét?

Jelöljük

- a specifications záradék invariánsát I_a -val,
- a representation záradékbeli invariánst I_c -vel.
- a specifications záradékban levő requirest β_{req} -val
- az initiallyt pedig β_{init} -tel.
- az inicializáló műveletet f_{init} –tel,
- a többi műveletet f -fel
- egy művelet végrehajtását $\{ \}$ jelekkel
- az f absztrakt műveletekhez tartozó elő- és utófeltételeket pre_a^f –vel és post_a^f –vel
- az f konkrét művelethez elő- és utófeltételeit pre_c^f –vel és post_c^f –vel

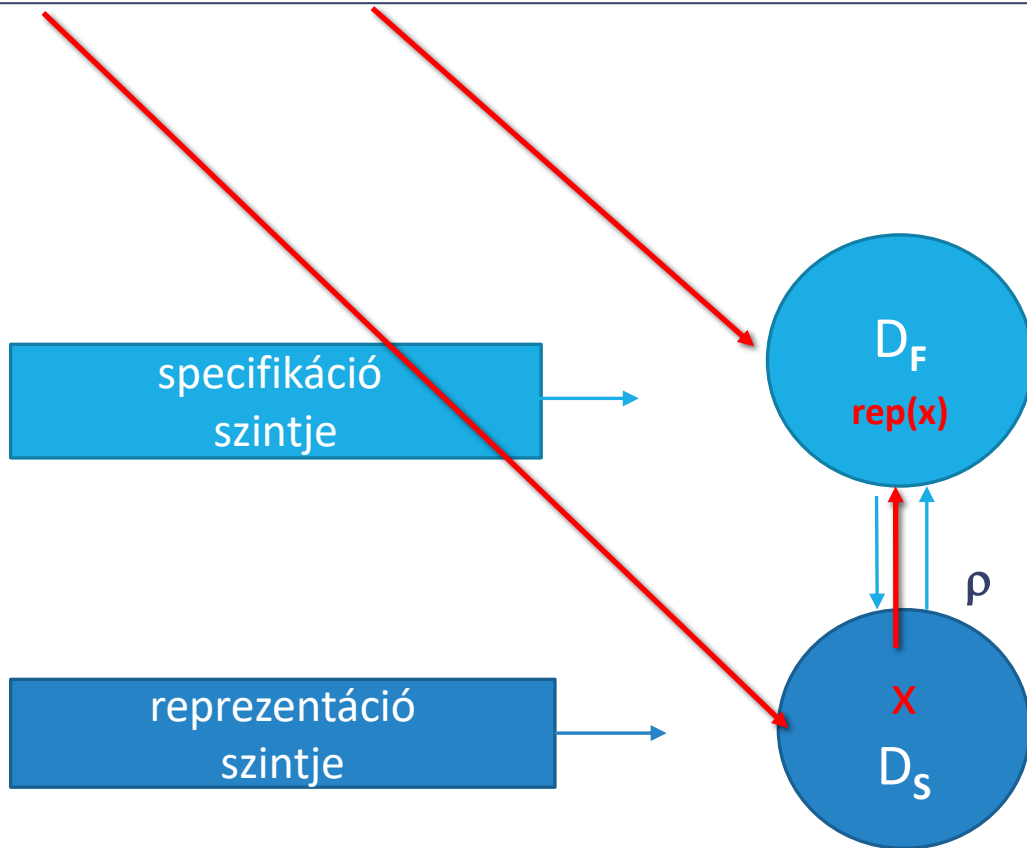
1. A reprezentáció helyességének ellenőrzése:

$$I_c(x) \Rightarrow I_a(\text{rep}(x))$$



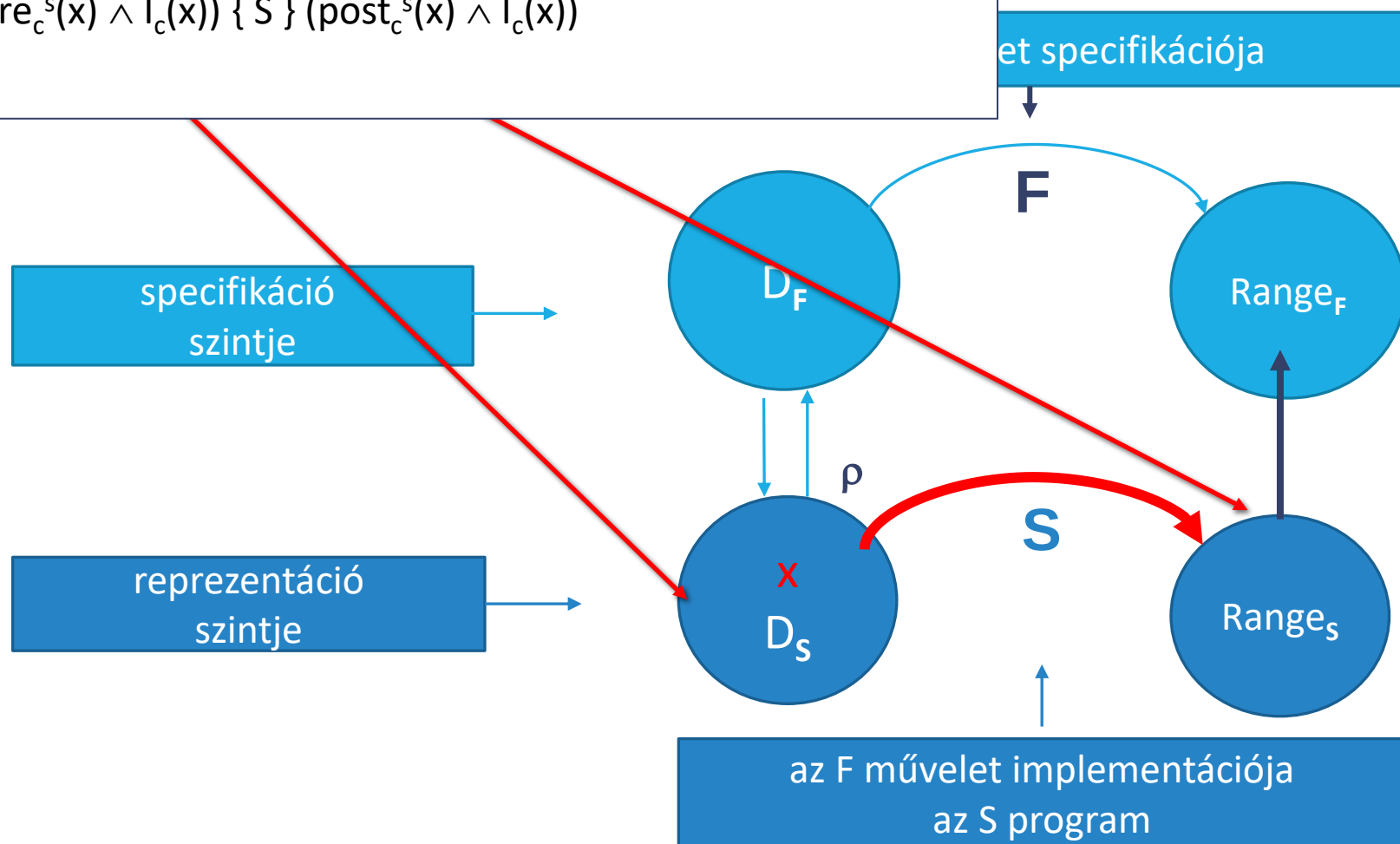
2. Az objektum inicializálásának ellenőrzése:

$$\beta_{\text{req}} \{f_{\text{init}}\} (\beta_{\text{init}} (\text{rep}(x)) \wedge I_c(x))$$



3. **Minden** f műveletre:

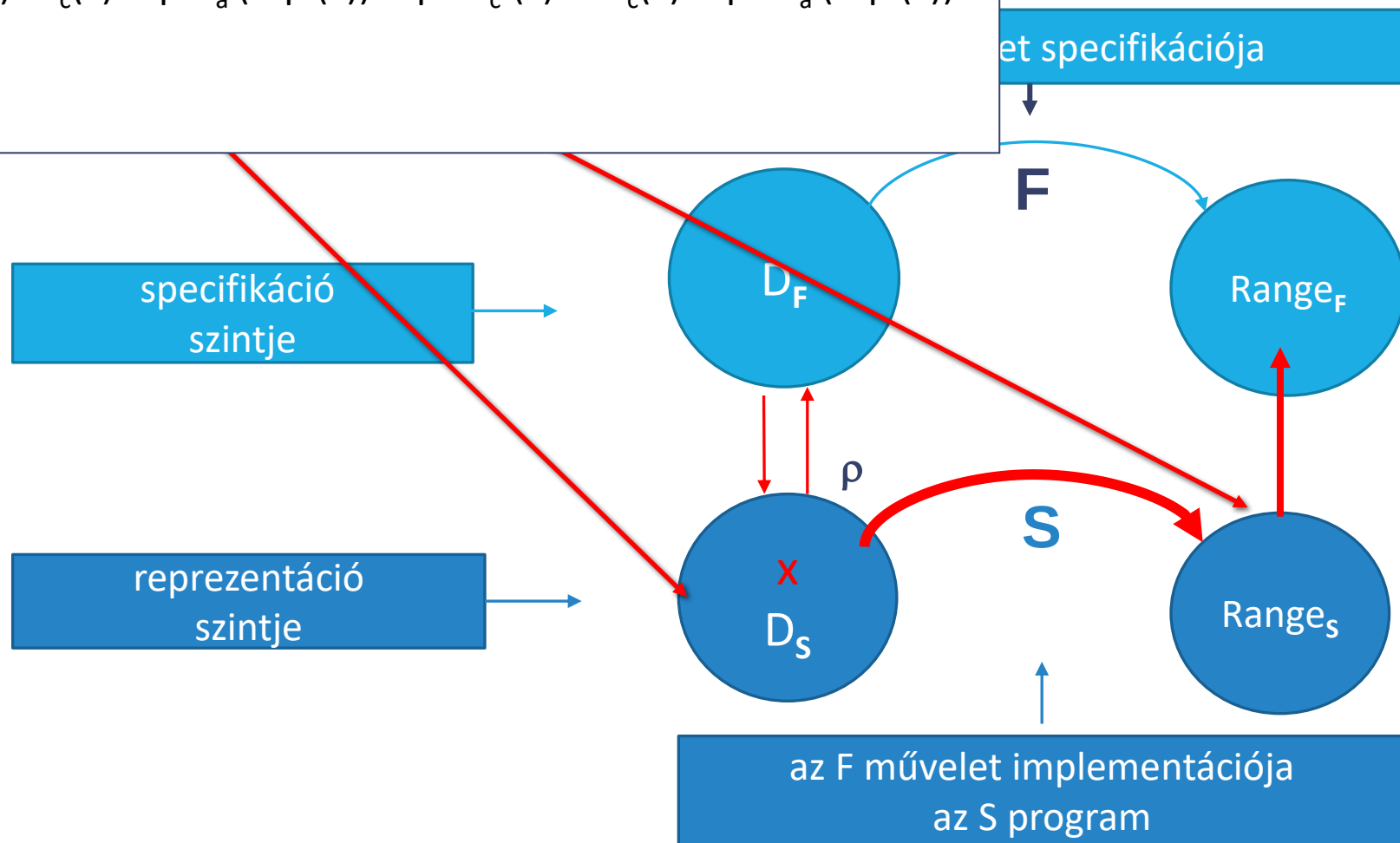
- a. A konkrét művelet (S)
helyességének ellenőrzése:
 $(\text{pre}_c^s(x) \wedge I_c(x)) \{ S \} (\text{post}_c^s(x) \wedge I_c(x))$



b. A konkrét és az absztrakt specifikáció közötti kapcsolat ellenőrzése:

$$(1) \quad I_c(x) \wedge \text{pre}_a^f(\text{rep}(x)) \Rightarrow \text{pre}_c^s(x)$$

$$(2) \quad I_c(x) \wedge \text{pre}_a^f(\text{rep}(x)) \wedge \text{post}_c^s(x) \Rightarrow I_c(x) \wedge \text{post}_a^f(\text{rep}(x))$$



„Design by Contract”

Programozás szerződéssel

- Bertrand Meyer „Object-Oriented Software Construction”
- 1986 óta

Jogok és kötelezettségek

- Az előfeltétel a megrendelő kötelezettsége
- Az utófeltétel a teljesítést vállalóé

Eiffel

Az Eiffel beépített nyelvi eszközökkel rendelkezik formális specifikációk megadására

- Az Eiffel futtató rendszer ellenőrizni is tudja, hogy az egyes programegységek nem sértik-e meg a specifikációt.

A típusok műveleteinek elő-utófeltételes specifikációja fontos szerepet játszik az Eiffel módszertanában:

- felhasználják a program tervezése során a követelmények pontosabb megfogalmazására, de teszteléskor és dokumentációs célokra is.

A nyelv tervezésekor ugyanakkor nem volt cél, hogy a fordítóprogram bizonyítsa a program helyességét, mert ez a gyakorlatban nem oldható meg.

Elő- utófeltételek

forth is

-- Move forward one position.

require

not_after: not after

do

... Some appropriate implementation

ensure

position = old position + 1

end - forth

put_i_th (v: like first; i: INTEGER) is

-- Put item v at i-th position.

require

index_large enough: i >= 1;

index_small_enough: i <= count;

deferred

ensure

not empty

end - put_i_th

Osztályinvariáns

deferred class CHAIN feature

...

invariant

-- Definitions:

empty = (count = 0);

off = ((position = 0) or (position = count + 1));

isfirst = (position = 1);

islast = (not empty and (position = count));

-- Axioms:

count >= 0;

position >= 0; position <= count + 1;

empty => (position = 0);

(not off) => (item = i_th (position));

-- Theorems:

(isfirst or islast) implies not empty

end -- class CHAIN

Az osztály konzisztens

Az osztály minden p konstruktor eljárására:

- $\{pre_p\} do_p \{INV_C\}$

Az osztály minden r metódusára, amit általában, vagy szelektíven exportál:

- $\{pre_r \wedge INV_C\} do_r \{post_r \wedge INV_C\}$

Megadás

Kvantorok megadására nincs lehetőség, de használhatjuk az `old` operátort:

- Az `old` operátorral egy objektum végrehajtandó metódusába való belépése előtti állapotát jelöljük;
Utófeltételben: $x = \text{old } x + 1$

Ciklushelyesség:

```
from -INIT
  go_before
invariant -- INV
  0 <= child_position; child_position <= arity + 1
variant -- VAR
  arity - child_position + 1
until -- EXIT
  child after or else (j = i)
loop -- BODY
  child_forth;
  if (sought = child) then
    j:=j+1
  end
end
end
```

Ciklusinvariáns és
terminátor függvény

Ciklushelyesség

A C osztály r metódusa akkor és csak akkor ciklushelyes, ha minden ciklusra:

1. {True} INIT {INV}
2. {True } INIT {VAR $\geq$ 0 }
3. {INV and then not EXIT} BODY { INV }
4. {INV and then not EXIT and then (VAR=v)}
BODY { 0 $\leq$ VAR<v }

Exception helyesség

Egy C osztály egy r rutinja exception-helyes akkor és csak akkor, ha a rescue blokkjának minden 'b' ágára a következő igaz:

- Ha b egy retry-jal végződik:
 - $\{ \text{true} \} b \{ \text{INV}_C \text{ and } \text{pre}_r \}$
- Ha b nem retry-jal végződik:
 - $\{ \text{true} \} b \{ \text{INV}_C \}$

Öröklődés és helyesség

Az öröklődés során az elő-utófeltételek újradeklarálhatóak, amelyre az Eiffel a **require else** ... alternatív előfeltételt és az **ensure then** ... extra utófeltételt adja.

- Legyenek $pre_1, \dots, pre_n$ az ősök előfeltételei, $post_1, \dots, post_n$ az ősök utófeltételei
- Minegyike **or else** kapcsolatban lesznek az előfeltételnél és **and then** kapcsolatban az utófeltételnél
- (az esetek többségében $n=1$).
- Ekkor a metódus előfeltétele:
 - alternatív előfeltétel **or else** pre_1 **or else** ... **or else** pre_n
utófeltétele:
 - alternatív utófeltétel **and then** $post_1$ **and then** ... **and then** $post_n$

Újraderfiniálás

A DbC szerint az újraderfiniálás

- **alszerződést** kötök a leszármazottal, amire a kliensek az őisével már kötöttek.
- Egy becsületes alvállalkozó legalább olyan jól teljesíti a követelményeket, mint az eredeti.
- Ez azt jelenti:
 - megtartja, vagy gyengíti az előfeltételt, vagyis nincs új követelménye a kliens felé,
 - megtartja vagy szűkíti az utófeltételt, vagyis az eredmény legalább olyan jó, mint az eredeti szerződésben volt.

Ellenőrzés

A program tetszőleges pontján elhelyezhetünk egy állítást, amelyet a futtató rendszer ellenőriz.

- Erre a **check** utasítás szolgál.

A környezet számos opciója segíti, hogy melyik állítást értékeljük ki. A lehetséges szintek:

- no - nem ellenőrzi az állításokat
- require - mindig az előfeltételeket ellenőrzi (alapértelmezett)
- ensure - előfeltételek és utófeltételek ellenőrzése
- invariant - az előzőeken túl az osztályinvariáns is
- loop - az előzőeken túl a ciklusok ellenőrzése
- all - az előzőeken túl a check utasítások ellenőrzése

Az egyes szintek jelentősége főleg a tesztelésnél van.

D nyelv

Assert a legegyszerűbb feltétel

- Ha nem teljesül, AssertionError-t dob

```
int i;  
Symbol s = null;  
for (i = 0; i < 100 ; i++)  
{  
    s = search(array[i]);  
    if (s != null) break;  
}  
assert (s != null); //megtaláltuk-e a keresett elemet
```

D nyelv

Elő és utófeltételek

`in { ... } out { ... } body { ... }`

- `in` és `out` elhagyható
- Ha csak `body` van, akkor a „`body`” tag is elhagyható
- `in`: a hívás előtt fut le
- `out`: a blokk elhagyásakor fut le, bárhogyan is lépünk ki belőle
- `body`: a függvény / blokk törzse
- Szabály: `in` és `out` bármilyen utasítást tartalmazhat, de a környezetet nem változtathatja meg
 - tipikusan `assert`ekkel van tele

D példa elő és utófeltételekre

```
long square_root(long x)
```

```
in
```

```
{
```

```
    assert(x >= 0);
```

```
}
```

```
out (result)
```

```
{
```

```
    assert((result * result) <= x &&  
           (result+1) * (result+1) > x);
```

```
}
```

```
body
```

```
{
```

```
    return cast(long)std.math.sqrt(cast(real)x);
```

```
}
```

D – osztály invariáns

Speciális tagfüggvény: invariant()

- Az osztály adattagjait nem módosíthatja
- A konstruktor elhagyásakor, a destruktor előtt, illetve minden egyes tagfüggvény hívása előtt és után lefut
- Az objektum helyességét ellenőrzi

```
class Date {  
    int day;  
    int hour;  
    invariant()  
    {  
        assert(1 <= day && day <= 31);  
        assert(0 <= hour && hour < 24);  
    }  
}
```

Oxygene

Object Pascal alapján

- Eiffel, C#, Java, F# hatások.

Elő és utófeltételek

- Az előfeltételt a `require` kulcsszó után kell megadni. Igaznak kell lennie a metódusba való belépéskor.
 - Például itt megkötéseket tehetünk a metódus paramétereire.
- Az utófeltételnél megadott feltételek az előtt értékelődnek ki, mielőtt a metódus befejeződne és a vezérlés az őt hívó blokkba térne vissza.
 - Az `ensure` kulcsszó után tehetjük
- Az elő és utófeltételeket egy osztály metódusánál lehet alkalmazni.

Oxygene – invariáns

invariants kulcsszó után lehet megadni.

- `public invariants`
 - Az összes publikus metódus végén leellenőrzi (az utófeltétel után).
- `private invariants`
 - Az összes metódus végén leellenőrzi

Oxygene – invariáns

type

MyClass = class;

public

... some methods or properties

public invariants

fField1 > 35;

SomeProperty = 0;

SomeBoolMethod() and not (fField2 = 5);

private invariants

fField > 0;

end;

Cobra

2009. február 29-én, MIT licenz alatt publikált általános célú, open source programozási nyelv, még jelentős fejlesztés alatt áll.

Chuck Esterbrook tervezte, a Python nyelvhez kapcsolódó fejlesztései miatt ismert.

Ebből következik, hogy a Cobra jelentősen épít a Python alapjaira

- A két nyelv szintaktikája majdnem teljesen megegyező
- Egyelőre halott ...

Cobra

A Cobra a contractok fogalmát átörököltette az Eiffel nyelvből.

- Megkövetelhetjük, hogy bizonyos elő-, utófeltételek és invariáns tulajdonságok teljesüljenek egy metódus végrehajtásakor.
 - require - előfeltételek
 - ensure - utófeltételek
- A feltételeket egymás alatt felsorolva, logikai kifejezések listájaként adhatjuk meg
 - invariant - osztályinvariáns
 - assert használható
- öröklődnek, a „szokásos” szabályok szerint

Cobra

```
class Person
  def drive(v as Vehicle)
    require
      not v.hasDriver
      v.isOperable
    ensure
      v.miles > old v.miles
    body
      ...
```

Cobra

```
class ContiguousList<of T>
  implements IList<of T>
    def insert(index as int, item as T)
      require
        index >= 0 and index < count
      ensure
        .count = old .count + 1
        this[index] is item
      body ...
```

Cobra – elő-utófeltétel öröklődés

```
class NonContiguousList<of T>
  inherits ContiguousList<of T>
  """
  Allows insertions past the end of the list.
  """
  def insert(index as int, item as T)
    or require index >= 0
    body ...
```

- utófeltételnél: **and ensure**

Cobra - invariáns

```
class Player
```

```
invariant
```

```
    .name.length
```

```
    .score >= 0
```

```
    .isAlive implies .health > 0
```

Java – assert

assert utasítás

Az utasítással különböző elő- és utófeltételeket, invariánsokat fogalmazhatunk meg.

Előnye a feltételes utasításokkal szemben: használata ki- és bekapcsolható.

Az utasításnak két alakja van:

- `assert logikai_kifejezés;`
- `assert logikai_kifejezés : üzenet;`

Java – assert

Például két változóról azt feltételezzük, hogy az egyik kisebb, mint a másik.

- Ekkor az utasítás így nézhet ki:
 - `assert x < y;`

Ha a feltétel nem teljesül, akkor az utasításnak ez a formája `AssertionError` hibát dob.

- Ha az üzenetet is megadjuk, akkor azt is kiírja.

Fontos szabály, hogy az `assert` utasítás feltételének nem szabad befolyásolnia az alkalmazás működését, ugyanis az előfeltevések használata kikapcsolható.

Nem érdemes például metódushívást szerepeltetni benne.

- Egyébként minden feltétel szóba jöhet.

Java – assert példa

Ellenőrzi, hogy a beolvasott érték 0 és 10 között van-e:

```
import java.util.Scanner;
public class AssertTest {
    public static void main( String args[] ) {
        Scanner input = new Scanner( System.in );
        System.out.print( "Enter a number between 0 and 10: " );
        int number = input.nextInt();
        // assert that the value is >= 0 and <=10
        assert ( number >= 0 && number <= 10 ):
            "bad number: " + number;
        System.out.printf( "You entered %d\n", number );
    } // end main
} // end class AssertTest
```

Enter a number between 0 and 10: 5

You entered 5

Java – JML

Java Modelling Language (JML)

- A JML egy formális viselkedési interfész-specifikációs nyelv.
 - A Java osztályok interfészének szokásos szintaktikus leírásán túl az egyes függvények viselkedését is specifikáljuk a rendszerben.
- A JML nyelvben a szerződéseket megfogalmazó feltételeket annotációs megjegyzések formájában helyezhetjük el a programkódban.
 - Ez lehet egy `/**-`gal kezdődő sor vagy egy `/*@ ... @*/` blokk.
- A kifejezésekben a Java jelölésrendszerét használhatjuk, kibővítve néhány speciális elemmel.
- Az elkészült kódot a JML saját fordítójával, a `jmlc`-vel kell lefordítanunk,
 - A kód fordítható marad az eredeti Java fordítóval is, mivel a JML kommentben szerepel
- A kontraktusok ellenőrzése kikapcsolható.

Java – JML

```
public class IMath {  
    /*@ requires (* x is positive *);  
    @ ensures \result >= 0 &&  
    @ (* \result is an approximation to square root of x *)  
    @ */  
    public static int isqrt(int x) { ... }  
}
```

Szintaxis	Jelentés
\result	Eredmény
$A \Rightarrow B$	A-ból következik B
$A \Leftarrow B$	B-ből következik A
$A \Leftrightarrow B$	A iff B
$A \Leftarrow! \Rightarrow B$	$\neg(A \Leftrightarrow B)$
\old(E)	E eredeti értéke

Minősítők

- Univerzális és egzisztenciális
 - \forall \exists
- Általános
 - \sum, \product, \min, \max
- Numerikus
 - \num_of

Java – JML

```
/*@ requires len >= 0
   @ ensures \result ==
   @         (\sum int j; 0 <= j && j < len; v[j])
   @*/
float sum (int v[], int len) {
    float s = 0.0;
    int i = 0;
    while (i < len) {
        s = s + v[i];
        i = i + 1;
    }
    return s;
}
```

Java – JML

```
byte[] a; /* The array a is sorted */
/*@ invariant
    (\forall int i;
        0 <= i && i < a.length-1;
        a[i] < a[i+1]);
@ */
```

Java – JML

Invariáns:

```
public class BankAccount {  
    ...  
    final static int MAX_BAL = 1000;  
    int balance;  
    /*@ invariant 0 <= balance &&  
        balance <= MAX_BAL;  
        @*/  
    ...  
}
```

Java – JML

```
public class BankAccount {  
    final static int MAX_BALANCE = 1000;  
    int balance;  
  
    // Elő és utófeltétel  
    /*@ requires amount >= 0;  
        ensures balance == \old(balance)+amount &&  
            \result == balance;  
    @*/  
    public int debit(int amount) {  
  
    }  
}
```

Java – cofoja

A Google –nál fejlesztették 2011-ben

Annotációkat lehet használni

- Típus invariáns – **Invariant** kulcsszó – minden publikus és csomagszintű láthatóságú metódus be- és kilépésnél, és a konstruktorok végén ellenőrzi, öröklődésnél **and** kapcsolat
- Előfeltételek – **Requires** kulcsszó - metódus belépésnél ellenőrzi, öröklődésnél **or** kapcsolat
- Utófeltételek – **Ensures** kulcsszó - metódus normál kilépésnél ellenőrzi, öröklődésnél **and** kapcsolat
- Kivételes utófeltételek - **ThrowEnsures Checked** - ha exception lépett fel, ezt ellenőrzi
- **old** és **result** lehetősége

Java – cofoja

```
@Invariant("size() >= 0")
interface Stack<T> {
    public int size();

    @Requires("size() >= 1")
    @Ensures({ "size() == old(size()) - 1",
               "result == old(peek())" })
    public T pop();
}
```

Java – valid4j

Külső könyvtár Java-hoz, amely a kódba beágyazva teszi lehetővé ellenőrzéseket

- Code by contract támogatása
- Előfeltételek definiálása
- Utófeltételek definiálása
- Kivételeket vált ki a feltételek megsértésekor

```
// Initialize members with valid arguments
this.message = require(message, containsString("Greetings"));
validate(v, containsString("Greetings"),
        otherwiseThrowing(MissingGreatException.class));
// Return valid results
return ensure(list, hasSize(greaterThan(1)));
```

.NET (C# és VB)

Code Contracts Library a .NET-ben

- <http://research.microsoft.com/en-us/projects/contracts/>

Előfeltételek: **Contract.Requires(...)**

- általában paraméterek ellenőrzésére
- az előfeltételben szereplő összes tagnak elérhetőnek kell lennie az adott helyen (különben az előfeltételt nem tudja értelmezni a hívó metódus)
- a megadott feltételeknek nem lehet mellékhatásuk.

.NET

Előfeltételek: `Contract.Requires(...)`

Példák:

- az `x` paraméter nem lehet null:

```
Contract.Requires ( x != null );
```

- ha az adott feltétel nem teljesül, milyen kivétel váltódjék ki:

```
Contract.Requires<ArgumentNullException>(x != null);
```

.NET

„Örökölt” követelmények:

- A legtöbb kódban a paraméterek ellenőrzésére if-then-throw szerkezet, ha ezek a metódus elején, át lehet alakítani előfeltételekké, de egy explicit contract metódushívás kell utána

- pl.: Requires, Ensures, EnsuresOnThrow vagy EndContractBlock

```
if ( x == null ) throw new ...
```

```
Contract.EndContractBlock();
```

```
// minden megelőző if-et előfeltételnek tekint
```

.NET

Utófeltételek

- Ellenőrzése az adott metódus végrehajtása után

- **Közönséges utófeltétel**

`Contract.Ensures()`

`Contract.Ensures( this .F > 0 );`

- **Kivételes utófeltételek:**

- Ha a metódus végrehajtása során kivétel váltódik ki, akkor is lehetőség van az utófeltétel ellenőrzésére.
- Ekkor a kivétel típusától (T) függően is lehet megadni feltételt:

`Contract.EnsuresOnThrow<T>( this.F > 0 );`

.NET

Speciális metódusok az utófeltételben:

- csak utófeltételek belsejében lehet használni
- `Contract.Result<T>()`
 - hivatkozik a T típusú visszatérési értékre (void típusú függvényeknél nem használható)
- `Contract.Ensures(0 < Contract.Result<int>());`

.NET

- `Contract.OldValue<T>(exp)`

az exp kifejezés metódus hívása előtt értékét adja vissza.

Megszorítások:

- az exp kifejezés nem tartalmazhat másik régi értéket lekérdező függvényt
- csak olyan kifejezésre hivatkozhat, aminek létezett értéke a metódus meghívása előtt.

- Példák lehetséges hibákra:

- A metódus visszatérési értékének a régi értékére nem lehet hivatkozni.

```
Contract.OldValue(Contract.Result<int>() + x) // ERROR
```

- A régi érték nem függhet a visszatérési értéktől.

```
Contract.ForAll(0, Contract.Result<int>(),  
    i => Contract.OldValue(xs[i]) > 3 ); // ERROR
```

.NET

`Contract.ValueAtReturn<T>(out T t)`

Az out paraméter értékének ellenőrzésére az utófeltételben.

```
public void OutParam(out int x){  
    Contract.Ensures(Contract.ValueAtReturn(out x) == 3);  
    x = 3;  
}
```

.NET

Invariánsok (Invariants)

- Az összes invariáns `void` típusú függvény kell legyen, ha az adott osztályból lehet származtatni, akkor a láthatóságnak `protected`-et kell megadni.
- A `[ContractInvariantMethod]` attribútummal jelöljük, hogy az adott metódus egy invariáns
- Az invariáns ellenőrzése az összes publikus metódus végrehajtása után megtörténik.
- Ha az invariánsban belül hivatkozunk az osztály egy másik publikus metódusára, akkor csak a legkülső függvényénél történik ellenőrzés.

.NET

[ContractInvariantMethod]

```
protected void ObjectInvariant()  
{  
    Contract.Invariant( this.y >= 0 );  
    Contract.Invariant( this.x > this.y );  
    ...  
}
```

.NET

Contract.Assert

- a program egy bizonyos pontján tudunk ellenőrizni:

```
Contract.Assert( this.privateField > 0 );  
Contract.Assert( this.x == 3,  
    "Why isn't the value of x 3?" );
```

Contract.Assume

- Egy feltevést ír le, működése megegyezik az Assert-tel futási időben, de itt fordítási idejű ellenőrzés is van!

```
Contract.Assume( this.privateField > 0 );  
Contract.Assume( this.x == 3,  
    "Static checker assumed this");
```

.NET

Contract.EndContractBlock

- ha az előfeltételek if-then-throw formában vannak leírva, akkor ez jelzi, hogy ezek előfeltételek, itt van az ellenőrző blokk vége:

```
if ( x == null ) throw
    new ArgumentNullException("x");
if ( y < 0 ) throw
    new ArgumentOutOfRangeException(...);
Contract.EndContractBlock( );
```

.NET

Contract.ForAll

- Ellenőrző ciklus, contract-on belül használható
- Két paraméteres változat:

```
public int Foo<T>(IEnumerable<T> xs){  
    Contract.Requires(  
        Contract.ForAll( xs, (T x) => x != null) );  
}
```

- Első paraméter egy kollekció
- A második egy predikátum
 - ha ez igaz a gyűjtemény minden elemére, akkor igazat ad vissza
 - ha van olyan elem, amire hamis, akkor megáll, és hamisat ad vissza.

.NET

- Három paraméteres változat:

```
public int[] Bar(){  
    Contract.Ensures(  
        Contract.ForAll(0, Contract.Result<int[]>().Length,  
            index => Contract.Result<int[]>()[index] > 0));  
}
```

- Első paraméter az alsó határ
- A második a felső határ
- A harmadik a predikátum, aminek van egy index argumentuma
 - A két határ között bejárja a predikátumban adott kollekció elemeit, ellenőrzi, hogy igaz-e mindegyikre.

.NET

Contract.Exists

- Ugyanolyan paramétereit vannak, mint a ForAll-nak
- Akkor tér vissza igaz értékkel, ha a kollekció legalább egy elemére teljesül az adott predikátum és hamis értékkel tér vissza, ha egyre sem.

.NET

Interfész contract-ok

- Itt nem írhatunk függvény törzset, ezért egy külön contract osztály kell, amit az interfésszel attribútumok kapcsolnak össze.

```
[ContractClass(typeof(IFooContract))]
```

```
interface IFoo {  
    int Count { get; }  
    void Put(int value );  
}
```

.NET

Interfész contract-ok

```
[ContractClassFor(typeof(IFoo))]  
abstract class IFooContract : IFoo {  
    int IFoo.Count {  
        get {  
            Contract.Ensures( 0 <= Contract.Result<int>() );  
            return default( int ); // dummy return  
            // lehet egy exc.-t is dobni itt  
        }  
    }  
    void IFoo.Put(int value){  
        Contract.Requires( 0 <= value );  
    }  
}
```

.NET

Absztrakt metódus szerződések

- itt se lehet függvény törzset írni, ezért itt is egy külön contract osztály kell, amit az absztrakt osztállyal attribútumok kapcsolnak össze:

```
[ContractClass(typeof(FooContract))]  
abstract class Foo {  
    public abstract int Count { get; }  
    public abstract void Put(int value );  
}
```

.NET

Absztrakt metódus szerződések (folyt.):

```
[ContractClassFor(typeof(Foo))]
```

```
abstract class FooContract : Foo {  
    public override int Count {  
        get {  
            Contract.Ensures( 0 <= Contract.Result<int>() );  
            return default( int ); // dummy return  
        }  
    }  
    public override void Put(int value){  
        Contract.Requires( 0 <= value );  
    }  
}
```

.NET

Contract metódusok túlterhelése:

- Mindegyik metódusnak lehet egy string típusú paramétere is.
- Ez kiíródik, ha a feltétel nem teljesül.
 - A string értékének fordítási időben ismertnek kell lennie.

```
Contract.Requires(x != null,  
    "If x is null, then the missiles are fired!");
```

Contract öröklődés

- Egy szerződés a típus altípusára is öröklődik.
- Az altípus előfeltételének gyengébbnek kell lennie ezért, ha az őstípusnál nincs megadva semmilyen előfeltétel, akkor az azt jelent, hogy az azonosan igaz az előfeltétele ezért az altípusokhoz nem lehet előfeltételt írni.
- Az utófeltételnél erősebbet kell / lehet megadni.

Ada 2012

Elő- és utófeltételek megadhatók

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

lehet az eredeti értékre is hivatkozni az utófeltételben:

- $\text{Post} \Rightarrow I = I'_{\text{Old}}$
- $A(I)'_{\text{Old}}$
- $A(I'_{\text{Old}})$

Ada 2012

Típusinvariáns

```
type Disc_Pt is private
```

```
    with Type_Invariant => Check_In(Disc_Pt);
```

```
type Stack is private
```

```
    with Type_Invariant => Is_Unduplicated(Stack);
```

Python – pycontracts

Python esetén több külső programkönyvtár is elérhető

- icontract
- PyContracts
- Decontractors
- dpcontracts
- zope.interface
- PyDBC
- Contracts for Python
- ...

Tervezték a „natív” támogatást is

- Egyelőre elhalasztották

Python – pycontracts

Példák megadásra:

```
@contract(a='int,>0', b='list[N],N>0', returns='list[N]')
```

```
def my_function(a, b):
```

```
@contract
```

```
def my_function(a : 'int,>0', b : 'list[N],N>0') -> 'list[N]':
```

```
    # Requires b to be a nonempty list, and the return
```

```
    # value to have the same length.
```

```
@contract
```

```
def my_function(a, b):
```

```
    """ Function description.
```

```
        :type a: int,>0
```

```
        :type b: list[N],N>0
```

```
        :rtype: list[N]
```

```
    """
```

Python – pycontracts

Több feltétel

```
@contract(prob='>=0,<=1')
```

```
def f(prob):
```

```
@contract(x='<0|>1')
```

```
def f(x):
```

```
@contract(x=int)
```

```
def f(x):
```

```
@contract(x='(int|float),>=0')
```

```
def f(x):
```

Python – pycontracts

Saját „contract” definiálása

- Dekorátor létrehozása

```
@new_contract
```

```
def even(x):
```

```
    if x % 2 != 0:
```

```
        msg = 'The number %s is not even.' % x
```

```
        raise ValueError(msg)
```

Kérdések

Az elő- és utófeltételes specifikációt alkalmazó programozási nyelvekkel kapcsolatos kérdések

- Alprogramoknak megadhatunk-e, és ha igen, milyen formában elő- és utófeltételeket?
- A típushelyesség ellenőrzéséhez megadhatunk-e specifikációs- és típusinvariánst?
- Kezeli-e a specifikációs elő- és utófeltételek és az implementált program elő- és utófeltételeinek különbségét, és így ellenőrzi-e a megfelelés helyességét?
- Vannak-e eszközei a ciklushelyesség ellenőrzéséhez?
- Exception-helyességet támogatja-e?
 - Exception kezelésnél legalább a típusinvariánst helyre kell állítani.
- Milyen az öröklődés és az elő- és utófeltételek, valamint a típusinvariáns kapcsolata?

„Programming methodology” kurzus MSc-n

FOLYTATÁSBAN