A Mercury nagyhatékonyságú LP megvalósítás

A fóliák szerzője: Benkő Tamás

Célok

- Nagybani programozás támogatása
- Produktivitás, megbízhatóság, hatékonyság növelése

Eszközök, elvek

- Teljesen deklaratív programozás
- Funkcionális elemek integrálása
- Hagyományos (Prolog) szintaxis megőrzése
- Típus, mód és determinizmus információk használata
- Szeparált fordítás támogatása
- Prologénál erősebb modul-rendszer
- Sztenderd könyvtár

Elérhetőség

- Fejlesztő (nyelv+implementáció): University of Melbourne
- `http://www.cs.mu.oz.au/mercury/`
- GPL

Mercury példaprogram

File-név illesztés

- A feladat: operációs rendszerek file-név-illesztéséhez hasonló funkció megvalósítása.

Adott minta és karaktersorozat illesztésekor

- A `?` egy tetszőleges karakterrel illeszthető.
- A `*` egy tetszőleges (esetleg üres) karakter-sorozattal illeszthető.
- A *c* karakter-pár a *c* karakterrel illeszthető, ha egy minta \-re végződik, az illesztés meghiúsul.
- Bármely más karakter csak önmagával illeszthető.

A Mercury program hívási formája:

`match Pattern1 Name Pattern2`

Itt a `Pattern1` és `Pattern2` mintákban a `*` és `?` azonos elrendezésben kell előforduljon.

A program funkciója

- a `Pattern1` mintára (az összes lehetséges módon) illeszti a `Name` nevet,
- a `*` és `?` karakterek helyébe kerülő szövegeket a `Pattern2` mintába behelyettesíti,
- és az így kapott neveket kiírja.

A file-név-illesztő Mercury program listája

A főprogram

```
:- module match.
/*-------------------------------------------------------*/
:- interface.

:- import_module io.
:- pred main(io__state::di, io__state::uo) is det. % kötelező

/*-------------------------------------------------------*/
:- implementation.
:- import_module list, std_util, string, char.

main -->
    command_line_arguments(Args),
    (   {Args = [P1,N1,P2]} ->
         {solutions(match(P1, N1, P2), Sols)},
         format("Pattern `%s' matches `%s' as `%s'\
 matches the following:\n\n",
                         [s(P1),       s(N1),  s(P2)]),
         write_list(Sols, "\n", write_string),
         write_string("\n*** No (more) solutions\n")
    ;   write_string("Usage: match <p1> <n1> <p2>\n")
    ).
```

Egyes könyvtári eljárások deklarációi

```
:- pred io__write_string(string, io__state, io__state).
:- mode io__write_string(in, di, uo) is det.
        % Writes a string to the current output stream.

:- pred io__write_list(list(T), string, pred(T, io__state, io__state),
        io__state, io__state).
:- mode io__write_list(in, in, pred(in, di, uo) is det, di, uo) is det.
        % io__write_list(List, Separator, OutputPred, IO0, IO)
        % applies OutputPred to each element of List, printing Separator
        % between each element. Outputs to the current output stream.

:- pred io__format(string, list(io__poly_type), io__state, io__state).
:- mode io__format(in, in, di, uo) is det.
        % io__format(FormatString, Arguments, IO0, IO).
        % Formats the specified arguments according to
        % the format string, using string__format, and
        % then writes the result to the current output stream.
        % (See the documentation of string__format for details.)
```


Példaprogram, folytatás

A program magja

```
:- pred match(string::in, string::in, string::in,
              string::out) is nondet. % szükséges
match(Pattern1, Name1, Pattern2, Name2) :-
    to_char_list(Pattern1, Ps1),
    to_char_list(Name1, Cs1),
    to_char_list(Pattern2, Ps2),
    match_list(Ps1, Cs1, L),
    match_list(Ps2, Cs2, L),
    from_char_list(Cs2, Name2).

:- type subst ---> any(list(char)) ; one(char).

:- pred match_list(list(char), list(char), list(subst)).
:- mode match_list(in, in, out) is nondet. % mindkét sor kell,,
:- mode match_list(in, out, in) is nondet. % vagy egyik se
match_list([], [], []).
match_list([?|Ps], [X|Cs], [one(X)|L]) :-
    match_list(Ps, Cs, L).
match_list([*|Ps], Cs, [any(Xs)|L]) :-
    append(Xs, Cs1, Cs),
    match_list(Ps, Cs1, L).
match_list([\, C|Ps],  [C|Cs], L) :-
    match_list(Ps, Cs, L).
match_list([C|Ps], [C|Cs], L) :-
    C \= (*), C \= ?, C \= (\),
    match_list(Ps, Cs, L).
```

A program fordítása, futása

```
> mmc match.m
> ./match '*b*' abbaba '* *'
Pattern `*b*' matches `abbaba' as `* *' matches the following:
a baba
ab aba
abba a
*** No (more) solutions
> ./match '**z?c' foozkc '|*|*|?'
Pattern '**z?c' matches 'foozkc' as '|*|*|?' matches the following:
|foo||k
|fo|o|k
|f|oo|k
|foo|k
*** No (more) solutions
```

Modul-rendszer

Támogatott tulajdonságok

- szeparált fordítás
- absztrakt típusok használata
- modulok egymásbaágyazása

Deklarációk

- modul kezdés: `:- module` ⟨modulename⟩`.`
- interfész: `:- interface.`
- megvalósítás: `:- implementation.`
- lezárás (opcionális): `:- end_module` ⟨modulename⟩`.`

Az interfész rész

- Minden szerepelhet, kivéve függvények, predikátumok és almodulok definíciója.
- Az itt szereplő dolgok fognak kilátszani a modulból.

Az implementációs rész

- Szerepelnie kell a függvények, predikátumok, absztrakt típusok és almodulok definíciójának.
- Az itt deklarált dolgok lokálisak a modulra.

Modul-rendszer, folytatás

Más modulok felhasználása

- `:- import_module` ⟨modules⟩`.`
  Ezután nem szükséges modulkvalifikáció.
- `:- use_module` ⟨modules⟩`.`
  Csak explicit modulkvalifikációval használhatjuk fel a benne levő dolgokat.

Modulkvalifikáció

- ⟨module⟩`:`⟨submodule⟩`:`...`:`⟨submodule⟩`:`⟨name⟩
- Egyelőre a `:` helyett a `__` javasolt, mert lehet, hogy később a `.` lesz a modulkvalifikátor és a `:` típuskvalifikátor.

Almodulok

- beágyazott almodulok: a főmodul fájljában definiált
- szeparált almodulok: külön fájlban definiált
- a jelenlegi implementációnál a beágyazott almodulok nem működnek

Típusok

A típusok fajtái

- primitív: `char`, `int`, `float`, `string`
- predikátum: `pred`, `pred(T)`, `pred(T1, T2)`, ...
- függvény: `(func) = T`, `func(T1) = T`, ...
- univerzális: `univ`
- „a világ állapota”: `io__state`
- felhasználó által bevezetett

Felhasználói típusok

- megkülönböztetett unió (SML: `datatype`)
- ekvivalencia (típusátnevezés) (SML: `type`)
- absztrakt adattípusik

Megkülönböztetett unió

Jellemzők

- Enumerációs és rekord típus
- lehet monomorf vagy polimorf

Enumeráció típus

```
:- type fruit ---> apple ; orange ; banana ; pear.
```

Rekord típus

```
:- type itree ---> empty ; leaf(int) ; branch(itree, itree).
```

Polimorfikus típus

```
:- type list(T) ---> [] ; [T|list(T)].
:- type pair(T1, T2) ---> T1 - T2.
```

A játékszabályok

- `:- type` ⟨típus⟩ `--->` ⟨törzs⟩`.`
- a ⟨törzs⟩ minden konstruktorában az argumentumok típusok vagy változók
- a ⟨törzs⟩ minden változójának szerepelnie kell ⟨típus⟩-ban
- ⟨típus⟩ változói különbözők
- a típusok között névekvivalencia van
- egy típusban nem fordulhat elő egynél többször azonos nevű és argumentumszámú konstruktor

Következmények

- egyszerű típusok általában „dobozolatlanul” implementálhatók
- „heterogén” kollekció esetében explicit csomagolásra van szükség

Más típusú típusmegadások

Ekvivalencia típus

- `:- type` ⟨típus⟩ `==` ⟨típus⟩`.`
- `:- type assoc_list(K, V) == list(pair(K, V)).`
- nem lehet ciklikus
- a jobb és a bal oldal ekvivalens

Absztrakt típus

- `:- type` ⟨típus⟩`.`
- `:- type t2(T1, T2).`
- a definíció el van rejtve az implementációs részben

A típusok használata

Predikátum-deklaráció

- A predikátumok és függvények argumentumainak meg kell mondani a típusát.
- `:- pred is_all_uppercase(string).`
- `:- func length(list(T)) = int.`

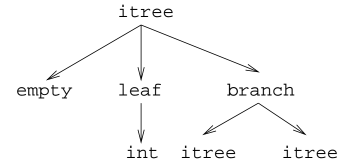
Módok, behelyettesítettség

Mód

- két behelyettesítettségi állapotból álló pár
- az első állapot arról szól, ahogy a paraméter bemegy, a második arról, ahogy kijön egy adott függvényből/predikátumból
- pl.: `out`: (szabad) változó megy be, tömör kifejezés jön ki

A behelyettesítettségi fa — példa

```
:- type itree ---> empty ; leaf(int) ; branch(itree, itree).
```



- Egy olyan fa, ahol a levelekben levő egészek behelyettesítetlenek:
  `:- inst bs = bound(empty; leaf(free); branch(bs,bs)).`
- Parametrizált `inst`-eket is csinálhatunk:

```
:- inst bs(Inst) = bound(empty ; leaf(Inst) ;
                         branch(bs(Inst),bs(Inst))).

:- inst listskel(Inst) = bound([] ; [Inst|listskel(Inst)]).}
```

Általánosan

- Az állapot leírásakor a típust tartalmazó („vagy”) csúcsokhoz rendelünk behelyettesítettségi állapotot.
- Deklarációban a `bound/1`, a `free/0` és a `ground/0` funktorokat használhatjuk.

Módok használata

Mód-deklaráció

- Módok definiálása:
  `:- mode` ⟨m⟩ `==` ⟨inst1⟩ `>>` ⟨inst2⟩`.`

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

- Módok átnevezése:
  `:- mode` ⟨m1⟩ `==` ⟨m2⟩`.`

```
:- mode (+) == in.
:- mode (-) == out.
```

- Parametrizált módok:
  `:- mode in(Inst) == Inst -> Inst.`
  `:- mode out(Inst) == free -> Inst.`

Predikátum-mód deklaráció

- Egy eljárás minden paraméteréről megmondjuk milyen módú.

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

- Egyetlen mód esetén összevonható a `pred` deklarációval.

```
:- pred append(list(T)::in, list(T)::in, list(T)::out).
```

- Függvényeknek is lehet több módja.
- Mercuryban egy adott predikátum egy adott módját nevezzük eljárásnak.

Módok: mire kell figyelni?

- `free` változókat még egymással sem lehet összekapcsolni,

```
:- mode append(in(listskel(free)),
               in(listskel(free)),
               out(listskel(free))).
```

  hibás!
- Ha egy predikátumnak nincs predikátum-mód deklarációja, akkor a fordító kitalálja az összes szükségeset (`--infer-modes` kapcsoló szükséges),
- de függvényeknél ilyenkor felteszi, hogy minden argumentuma `in` és az eredménye `out`.
- A fordító átrendezi a hívásokat, hogy a mód korlátokat kielégítse: ha ez nem megy, hibát jelez. (Jobbrekurzió! Lásd a `match_list/3` `append/3` hívását!)
- A megadottnál „jobban” behelyettesített argumentumokat egyesítésekkel kiküszöböli a fordító. Ezeket a módokat le se kell írni (de érdemes lehet). Példa: `:- mode append(in, out, in).` a szétszedő `append`-et fogja használni, ami nem hatékony:

```
append([1,2,3], X, [1,2,3,4,5])
---->                   append(U, X, [1,2,3,4,5]), U = [1,2,3].
```

- A jelenlegi implementáció nem kezeli a részlegesen behelyettesített adatokat.

Determinizmus

Determinizmus kategóriák
Minden predikátum minden módjára (azaz minden eljárásra) megadjuk, hogy hányféleképpen sikerülhet, és hogy meghiúsulhat-e.

A kategóriák nevei

meghiúsulás\megoldások	0	1	> 1
nem	`erroneous`	`det`	`multi`
igen	`failure`	`semidet`	`nondet`

A determinizmus-deklaráció

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet.
```

Összevont predikátum-, mód- és determinizmus-deklaráció

```
:- pred p(int::in) is det.
p(_).
```

„Egzotikus" determinizmusok

- `failure` determinizmusú a `fail/0`
- `erroneous` determinizmusú a `require__error/1`

Függvények determinizmusa

- Ha minden argumentuma bemenő, akkor a determinizmusa csak `det`, `semidet`, `erroneous` vagy `failure` lehet.
- Ha nem így lenne, akkor az matematikai értelemben nem lenne függvény.
- Pl. `between(in, in, out)` nem írható függvényalakban.

Példák

Helyesek-e?

```
:- type fruit ---> banana ; orange ; lemon ; grape.
:- type ice_cream ---> lemon ; banana ; orange.
:- type unsi ---> z ; s(unsi).
```

Milyen módjai vannak és milyen a determinizmusa?

```
:- pred make_ice_cream(fruit, ice_cream).
make_ice_cream(lemon,  lemon).
make_ice_cream(orange, lemon).
make_ice_cream(banana, banana).

:- func factorial(int) = int.
factorial(N) = F :-
         (   N = 0 -> F = 1
         ;   N > 0 -> F = factorial(N-1)*N
         ;   require__error("out of domain")
         ).

:- pred even(num).
even(z).
even(s(N)) :-
         odd(N).

:- pred odd(num).
odd(s(N)) :-
         even(N).
```


Magasabbrendű eljárások

Részlegesen paraméterezett eljárások

- segédeszközök: `call/2`, `call/3`, ... eljárások
- a `call/<I>` eljárások Mercuryban beépítettek

A `call/4` eljárás Prolog definíciója

```
% Pred az A, B és C utolsó argumentumokkal
% meghívva igaz.
call(Pred, A, B, C) :-
    Pred =.. FArgs,
    append(FArgs, [A,B,C], FArgs3),
    Pred3 =.. FArgs3, call(Pred3).
```

Példa: a `map` eljárás definíciója

```
% map(Pred, Xs, Ys): Az Xs lista elemeire
% a Pred transzformációt alkalmazva kapjuk az Ys listát.
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(in, out) is semidet, in, out) is semidet.
:- mode map(pred(in, out) is multi, in, out) is multi.
:- mode map(pred(in, out) is nondet, in, out) is nondet.
:- mode map(pred(in, in) is semidet, in, in) is semidet.
map(P, [H|T], [X|L]) :-
    call(P, H, X),
    map(P, T, L).
map(_, [], []).

:- import_module int.

:- pred negyzet(int::in, int::out) is det.
negyzet(X, X*X).

:- pred p(list(int)::out) is det.
p(L) :-
    map(negyzet, [1,2,3,4], L).

:- pred p1(list(int)::out) is det.
p1(L) :-
    map((pred(X::in, Y::out) is det :- Y = X*X), [1,2,3,4], L).
```


Magasabbrendű kifejezések létrehozása — példák

Magasabbrendű eljárások

- Tegyük fel, hogy létezik egy `sum/2` eljárás:

  ```
  :- pred sum(list(int)::in, int::out) is det.
  ```

- Ekkor eljárás-értéket létrehozhatunk
  - λ-kifejezéssel:

    ```
    X = (pred(Lst::in, Len::out) is det :- sum(Lst, Len))
    ```

  - az eljárás nevét használva (a nevezett dolognak csak egyféle módja lehet és nem lehet 0 aritású függvény):

    ```
    Y = sum
    ```

- `X` és `Y` típusa: `pred(list(int), int)`

Magasabbrendű függvények

- Tegyük fel, hogy létezik egy `mult_vec/2` függvény:

  ```
  :- func mult_vec(int, list(int)) = list(int).
  ```

- Ekkor függvény-értéket létrehozhatunk
  - λ-kifejezéssel:

    ```
    X = (func(N, Lst) = NLst :- NLst = mult_vec(N, Lst))
    Y = (func(N::in, Lst::in) = (NLst::out) is det
            :- NLst = mult_vec(N, Lst))
    ```

  - a függvény nevét használva:

    ```
    Z = mult_vec
    ```

Többargumentumú megasabbrendű kifejezések (currying)

Eljárások és függvények

- `Sum123 = sum([1,2,3])`: `Sum123` típusa `pred(int)`
- `Double = mult_vec(2)`: `Double` típusa `func(list(int)) = list(int)`

DCG

- Külön szintaxis az olyan eljárásokra, amelyek egy akkumulátorpárt használnak
- Példa (típusa `pred(list(string), int, io__state, io__state)`):

```
Pred = (pred(Strings::in, Num::out, di, uo) is det -->
    io__write_string("The strings are: "),
    { list__length(Strings, Num) },
    io__write_strings(Strings),
    io__nl
)
```

Amire figyelni kell

- beépített nyelvi konstrukciókat nem lehet „curryzni”
- ilyenek pl.: `=`, `\=`, `call`, `apply`
- `list__filter([1,2,3], \=(2), List)` helyett:
  `list__filter([1,2,3], (pred(X::in) is semidet :- X \= 2), List)`

Magasabbrendű eljárások és függvények meghívása

- `call(Closure, Arg`$_1$`, ..., Arg`$_n$`)`, $n \geq 0$
- példa: `solutions(match(P1, N1, P2), Sols)`
- `apply(Closure2, Arg`$_1$`, ..., Arg`$_n$`)`, $n \geq 0$
- példa: `List = apply(Double, [1,2,3])`

Magasabbrendű módok

Mód és determinizmus

- A magasabbrendű kifejezések determinizmusa a módjuk része (és nem a típusuké).
- Például:

```
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
```

Beépített behelyettesítettségek

- Eljárások:
  `pred(`$\langle$mode$_1\rangle$`, ..., `$\langle$mode$_n\rangle$`) is` $\langle$determinism$\rangle$, ahol $n \geq 0$
- Függvények:
  `(func) =` $\langle$mode$\rangle$ `is` $\langle$determinism$\rangle$
  `func(`$\langle$mode$_1\rangle$`, ..., `$\langle$mode$_n\rangle$`) =` $\langle$mode$\rangle$ `is` $\langle$determinism$\rangle$, ahol $n > 0$

Beépített módok

- A nevük megegyezik a behelyettesítettségek nevével, és a pár mindkét tagja ugyanolyan, a névnek megfelelő behelyettesítettségű.
- Egy lehetséges definíció lenne:

```
:- mode (pred(Inst) is Det) == in(pred(Inst) is Det).
```

Amire figyelni kell

- Magasabbrendű kimenő paraméter:

```
:- pred foo(pred(int)).
:- mode foo(free -> pred(out) is det) is det.
foo(sum([1,2,3])).
```

- Magasabbrendű kifejezések nem egyesíthetők:
  `foo((pred(X::out) is det :- X = 6))` hibás.

Problémák a determinizmussal

- `det` és `semidet` módú eljárásokból nem hívható `nondet` vagy `multi` eljárás
- például a `main/2` eljárás `det` módú

Megoldások

- az összes megoldást megkeressük: `std_util__solutions/2`
- csak egy megoldást akarunk (és nem érdekes melyik)
  - ha az eljárás kimenő változóit nem használjuk fel, akkor az első utáni megoldásokat levágja a rendszer: `member(1, [1,1])`
  - kihasználjuk, hogy sosem fogunk egynél több megoldást keresni (committed choice nondeterminism): `cc_nondet`, `cc_multi` determinizmus
- (néhány megoldást keresünk meg: `std_util__do_while/4`)

Amire még nincs igazi megoldás

- meg akarunk hívni egy eljárást, amelynek minden megoldása ekvivalens
- tervezett megoldás: `unique [X] goal(X)`
- egyelőre a C interfésszel kell trükközni

Problémák a determinizmussal, példa

Feladat

1. Soroljuk fel egy halmaz összes részhalmazát!
2. Minden megoldást pontosan egyszer adjunk ki!

```
:- module resze.

:- interface.
:- import_module io.

:- pred main(io__state::di, io__state::uo) is cc_multi.

:- implementation.
:- import_module int, set, list, std_util.

main -->
        read_int_listset(L, S),
        io__write_string("Set version:\n"),
        {std_util__unsorted_solutions(resze(S), P)},
        io__write_list(P, " ", io__write),
        io__write_string("\n\nList version:\n"),
        {std_util__unsorted_solutions(lresze(L), PL)},
        io__write_list(PL, " ", io__write), io__nl.

:- pred read_int_listset(list(int)::out, set(int)::out,
                         io__state::di, io__state::uo) is det.
read_int_listset(L, S) -->
        io__read(R),
        {   R = ok(L0) ->
        ->  L = L0,
            set__list_to_set(L, S)
        ;   set__init(S),  % S := üres halmaz
            L = []
        }.
```

Problémák a determinizmussal, folytatás

1. megoldás: `set` absztrakt adattípussal
A `set__member/2` felsoroló jellege miatt nem teljesíti a 2. feltételt.

```
:- pred resze(set(T)::in, set(T)::out) is multi.
resze(A, B) :-
        set__init(Fix),  % Fix := üres halmaz
        resze(A, B, Fix).

:- pred resze(set(T)::in, set(T)::out, set(T)::in) is multi.
resze(A, B, Fix) :-
        (   set__member(X, A)
        ->  set__delete(A, X, A1),
            (   resze(A1, B, Fix)
            ;   resze(A1, B, set__insert(Fix, X))
            )
        ;   B = Fix
        ).
```

2. megoldás: `list` adattípussal
A lista fejének levágása (szemi)determinisztikus, így teljesül a 2. feltétel.

```
:- pred lresze(list(T)::in, list(T)::out) is multi.
lresze(A, B) :-
        lresze(A, B, []).

:- pred lresze(list(T)::in, list(T)::out, list(T)::in) is multi.
lresze(A, B, Fix) :-
        (   A = [X|A1],
            (   lresze(A1, B, Fix)
            ;   lresze(A1, B, [X|Fix])
            )
        ;   A = [], B = Fix
        ).
```

Példafutás

```
> ./resze
[1, 2].
Set version:
[1, 2] [2] [1] [] [1, 2] [1] [2] []

List version:
[2, 1] [1] [2] []
>
```


Committed choice nondeterminism

Használat

- olyan helyeken használhatjuk, ahol biztosan nem lesz szükségünk több megoldásra
- `cc_multi` a `multi` helyett
- `cc_nondet` a `nondet` helyett
- két predikátummód-deklaráció különbözhet csak a `cc`-s mivoltukban

```
:- mode append(out, out, in) is multi.
:- mode append(out, out, in) is cc_multi.
```

- I/O műveletek csak `det` és `cc_multi` eljárásokban lehetségesek

Egy `cc_multi`-s példa

```
:- module queens.
:- interface.
:- import_module list, int, io.
:- pred main(state::di, io__state::uo) is cc_multi.
:- implementation.

main -->
        (   {queen([1,2,3,4,5,6,7,8], Out)} -> write(Out)
        ;   write_string("No solution")
        ), nl.

:- pred queen(list(int)::in, list(int)::out) is nondet.
queen(Data, Out) :-
        perm(Data, Out), safe(Out).

:- pred safe(list(int)::in) is semidet.
safe([]).
safe([N|L]) :-
        nodiag(N, 1, L), safe(L).

:- pred nodiag(int::in, int::in, list(int)::in) is semidet.
nodiag(_, _, []).
nodiag(B, D, [N|L]) :-
        D \= N-B, D \= B-N, nodiag(B, D+1, L).
```


Egyszeres hivatkozású (unique) módok

Jellemzők

- Az adott paraméterre csak egy referencia lehet.
- A referencia megszűntével a memória felszabadítható vagy újrahasznosítható.
- Segítségével destruktív frissítés valósítható meg.
- Ezt használja pl. az `io` könyvtár is.

Új behelyettesítettségek

- `unique`: olyan, mint `ground`, de csak egyszeres hivatkozás lehet
- `unique(...)`: olyan, mint `bound(...)`, de csak egyszeres hivatkozás lehet
- `dead`: nincs rá több hivatkozás

Sztenderd módok

- `:- mode uo == free >> unique.`
- `:- mode ui == unique >> unique.`
- `:- mode di == unique >> dead.`

A jelenlegi implementáció korlátai

- csak a legfelső szinten megengedett a `unique` behelyettesítettség
- a memória újrahasznosítása csak az `io` és az `array` könyvtárakban működik