A MAUDE specifikációs nyelv

Lukácsy Gergely

stilgar@cs.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem

2003. november 3.

A Maude dióhéjban

Főbb jellemzők

- a Maude egy deklaratív nyelv
- Kétféle nyelv létezik a `Core Maude` és a `Full Maude`. Az alapnyelvet C++-ban implementálták, a kiegészítést már Maude-ban.
- az OBJ nyelvből fejlődött ki, az OBJ-t 1976-ban kezdték fejleszteni

Alkalmazási területek

- modell ellenőrzés
- konkurrens és elosztott rendszerek
- protokollok specifikációja

Előadás menete

- Bevezetés, modulok
- Alap nyelvi konstrukciók
  - kategóriák
  - operátorok
    - precedencia, gyűjtő minták
    - operátor túlterhelés
  - változók, termek
  - hibakategóriák
  - könyvtári modulok
- A funkcionális modulok
  - egyenletek
  - operátor attribútumok
  - elemségi axiómák
  - operátor kiértékelési stratégiák

- táblázás Maude-ban
- hibakezelés Maude-ban

- A rendszer modulok
  - átírási szabályok
  - átírási stratégiák
  - szabályok fagyasztása
- Full Maude rendszer - gyors áttekintés
  - objektum-orientált modulok
- Ami kimaradt
  - szabályok fagyasztása
  - meta programozás Maude-ban, reflekció
  - model ellenőrzés
- összefoglalás

Magyarázat nélkül

```
fmod PEANO-NAT is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor iter] .
    op _+_ : Nat Nat -> Nat .

    vars M N : Nat .
    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

Bevezetés, modulok

Maude-ban a programok alapegysége a **modul**. Egy Maude modul egy matematikai értelemben vett *algebrát* definiál, azaz egy alaphalmazt, annak részhalmazait (típusok) és a rajtuk értelmezett operátorokat.

- **funkcionális modulok** : egyenletekel megadott funkcionális program, általános alakja:

  ```
  fmod MODULNEV is
    ...
  endfm
  ```

- **rendszer modulok** : átírási szabályokkal bővített funkcionális modul

  ```
  mod MODULNEV is
    ...
  endm
  ```

- **objektum-orientált modulok** :

  ```
  omod MODULNEV is ... endom
  ```

Alap nyelvi konstrukciók, kategóriák

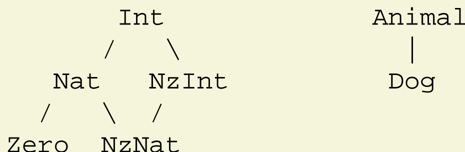
- A Maude-ban (lényegében) nincsenek beépített típusok, még számaink sincsenek. Saját típusokat, kategóriákat a `sort` kulcsszóval definiálhatunk

```
sort Nat .
sorts NzNat Zero .
```

- az alkategória reláció részleges rendezést definiál a kategóriák között, nem szabad köröket létrehozni (a rendszer ellenőrzi)

```
subsorts NzNat Zero < Nat .
```

- a részleges rendezés összefüggő komponenseket hoz létre, minden komponensben van egy (vagy több) **főkategória**



Operátorok

Operátorokat Maude-ban az `op` kulcsszóval definiálhatunk, ahol megadjuk az operátor nevét, argumentumainak és az eredményének a típusát.

```
op + : Nat Nat -> Nat .
ops * + : Nat Nat -> Nat .
```

- a fenti példákban *prefix* jelölést alkalmaztunk, lehetőség van azonban az ún. **mixfix** jelölésre is

  ```
  op _+_ : Nat Nat -> Nat .
  op s_ : Nat -> NzNat .
  op AddMax_+_to_ : Int Int List -> List .
  ```

Ha az argumentumlista üres, az operátort **konstansnak** hívjuk.

```
op 0 : -> Nat .
ops piros sárga zöld : -> Color .
```

A mixfixként jelölt operátorokat is alkalmazhatjuk prefix pozícióban, az alábbiak közül mindkettő helyes

```
_+_(0,0)
0 + 0
```

Operátorok - folytatás

A mixfix forma egy speciális esete az, amikor az aláhúzásjelek között nincsen semmi

```
sort NatSeq .
subsort Nat < NatSeq .
op __ : NatSeq NatSeq -> NatSeq .
```

Ekkor a…

```
0 s(0) s(s(0))
```

…egy `NatSeq`-et jelöl.

Operátor precedencia

Az operátorokhoz a programozó *precedenciát* rendelhet (a precedencia természetes szám, 0 jelöli a legerősebben kötő operátort). Amennyiben ezt nem teszi meg, akkor a Maude jól definiált szabályok szerint automatikusan kioszt ilyeneket. Példák:

- minden mixfix operátor, ami mással kezdődik **és** végződik, mint aláhúzás 0 precedenciát kap; ilyenre példa: `(_)`, `<_:_|_>` , `if_then_else_fi`
- minden egyargumentumú mixfix operátor, ami aláhúzással kezdődik **vagy** végződik 15-ös precedenciájú; példa: `not_`, `_!`.
- ...

A programozó a `prec` operátor jelző segítségével definiálhat precedenciát.

```
op _+_ : Nat Nat -> Nat [prec 35] .
op _*_ : Nat Nat -> Nat [prec 25] .
```

Ekkor biztos lehet benne, hogy a `1 + 2 * 3` mindig `1 + (2 * 3)`-ként értékelődik ki. Alapesetként mindkét operátor a 41 precedenciát kapná és ilyenkor nem egyértelmű az elemzés.

Fontos, hogy a precedencia zárójelekkel mindig felülbírálható.

Operátor precedencia - folytatás

A `prec` használata önmagában nyilvánvalóan nem elegendő, hiszen egy az alábbi kifejezés még mindig nem egyértelmű (megjelölhetnénk persze az operátort az `assoc` jelzővel)

```
1 + 2 + 3
```

Mi a helyzet akkor, ha nem asszociatív az operátor, mint például a `_vs_`? Mit jelent `sampras vs agassi vs moya`, ha adottak az alábbiak:

```
op _vs_ : Player Player -> Player [comm] .
ops sampras agassi moya : -> Player .

eq moya vs agassi = moya .
eq agassi vs sampras = agassi .
eq sampras vs roddick = sampras .
```

A Maude megoldása a **gyűjtő minták** bevezetése. Minden operátor argumentumhoz tartozhat egy jel a következők közül: `e`, `E`, `&`

- `e`: az argumentumhelyen (az operátornál) csak **kisebb** precedenciaértékű term állhat
- `E`: az argumentumhelyen **nem nagyobb** precedenciaértékű term állhat
- `&`: nincsen semmilyen megkötés (ez az alapértelmezett)

Gyűjtő minták - folytatás

Az operátorokhoz az alapértelmezett precedencián kívül tartozik alapértelmezett *gyűjtő minta* is. Az alábbi esetben ez az `(E E)`:

```
op _+_ : Nat Nat -> Nat [prec 35] .
op _*_ : Nat Nat -> Nat [prec 25] .
```

Ez azt jelenti, hogy az `(1 + 2) * 3` elemzés nem helyes, hiszen `1 + 2` precedenciája 35 (egy term precedenciája a legkülső operátor precedenciája) `_*_` bal argumentuma azonban nem lehet nagyobb precedenciájú, mint 25. Fontos látni tehát, hogy a precedencia a *gyűjtő minták*-kal együtt működik csak.

Amennyiben átállítjuk az alapértelmezett *gyűjtő mintát* `(E e)`-re...

```
op _+_ : Nat Nat -> Nat [prec 35 gather (E e)] .
```

...az egyetlen helyes elemzése az `1 + 2 + 3`-nak az `(1 + 2) + 3`, azaz bal asszociatívvá tettük `_+_`-t.

Megjegyzés: `1 + 2`, `sampras vs agassi` mindig elemezhető, mert a konstansok precedenciája 0. Hasonló igaz a `(_)` operátorra is.

Operátor túlterhelés

Maude-ban lehetőségünk van arra, hogy különböző operátorokat azonos néven definiáljunk.

```
op _+_ : Int Int -> Int
op _+_ : Nat Nat -> Nat
op _+_ : NzNat Nat -> NzNat
op _+_ : String String -> String
```

Az operátor túlterhelésnek két fajtáját különböztetjük meg

- **altípus túlterhelés**: példánkban ilyen a második és a harmadik definíció (feltesszük, hogy `Nat < Int`); ilyenkor az új operátordefiníció a régi speciális esete; arra is lehetőség van, hogy *konstanst* terheljünk túl

  ```
  op 0 : -> Nat .
  sorts NzNat Zero .
  subsorts NzNat Zero < Nat .
  op 0 : -> Zero .
  ```

- **ad-hoc túlterhelés**: példánkban ilyen az utolsó; ad-hoc esetben nem lehet konstanst túlterhelni (bár erre is van kerülőút)

Változók, Termek

A Maude-ban egy változó egy adott kategóriához (vagy hibakategóriához) kötődik, a kategória egy tetszőleges értékét veheti fel.

```
var x : Nat .
vars c1 c2 c3 : Color .
```

- egy változónak Maude-ban sosincs *konkrét* értéke, nincsen értékadás, nem tárolhatunk bennük részeredményt, amit később felhasználunk stb.
- jelentőségük az egyenleteknél és az átírási szabályoknál van

Változókat a fentiektől eltárően *on-the-fly* módon is definiálhatjuk.

Egy Maude **term**(**kifejezés**) változók, konstansok, vagy operátor alkalmazások. Az alábbiak Maude kifejezések

```
s_(0)
s 0
0
0 + 0
N
```

Hibakategóriák

Maude-ban minden kategóriához implicit módon tartozik egy **hibakategória** (*kind*). Az ugyanazon összefüggő komponensben lévő kategóriákhoz ugyanazon hibakategória rendelődik hozzá (valójában a kategóriák ekvivalenciaosztályokba sorolódnak és egy-egy ekvivalenciaosztályhoz tartozik egy *kind*).

- a *kind*-ok nincsenek explicit módon elnevezve, hivatkozni rájuk `[ ]`-be tett kategórianévvel, illetve nevekkel lehet
- például `[Nat]`, `[NzNat]`, `[Int]` vagy `[Nat, Int]` ugyanazon *kind*-ot reprezentálja (amennyiben ezen kategóriák egy komponensben vannak)
- a Maude egy kanonikus alakot használ, a `[ ]`-ban az összefüggő komponens főtípusa(i) szerepelnek

Például definiáljunk egy speciális *előző* tulajdonságot a Peano világban

```
op p : NzNat -> Nat .
```

Ekkor, a `p(0)` kifejezést elemzi a Maude, de a kifejezés hibakategóriáját [Int]-re állítja. Az olyan kifejezések, melyeknek nincsen kategóriája, csak hibakategóriája **hiba kifejezésnek** hívjuk.

Hibakategóriák - folytatás

Egy másik példa állatok szaporodását írja le.

```
sort Dog .
ops dalmatian pitbull schnauzer : -> Dog .
op breed : Dog Dog -> Dog [ctor] .
```

Ekkor `breed(pitbull,breed(dalmatian,schnauzer))` kifejezés egy kutyust jelent. Ha azonban bevezetjük a pingvineket az alábbi módon,

```
sort Animal .
subsort Dog Animal .
op penguin : -> Animal .
```

akkor `breed(schnauzer,penguin)` egy hiba kifejezés, `[Animal]` hibakategóriával.

Fontos, hogy amennyiben a szóban forgó argumentum esetében az elvárt és a kapott kategória **külön** összefüggő komponensben van, akkor a kifejezést nem is elemzi a Maude:

```
sort Vegetable .
op carrot : -> Vegetable .
```

A `breed(schnauzer,carrot)` egyáltalán nem olvasódik be.

Hibakategóriák - összetett példa

```
sorts Plea Inquiry Assertion Question Speech .
sort LogEntry .
subsorts Inquiry < Question < Plea .
subsorts Assertion Question < Speech .

op respond : Question -> Speech .
op record : Speech Speech -> LogEntry .

ops p1 p2 : -> Plea .
...
```

Melyek elemződnek és mi lesz a kategóriájuk vagy hibakategóriájuk?

```
respond(q1)                      Plea    Speech
record(q1,i1)                       \     /    \
respond(record(q1,i1))             Question   Assertion       LogEntry
record(q1,respond(q1))                /
respond(p1)                         Inquiry
record(p1,p2)
record(respond(i1),s1)
record(i1,respond(i1))
respond(s1)
```

Könyvtári modulok

A `prelude.maude` állomány a Maude rendszer indulásakor automatikusan betöltődik. Ez egy (majdnem) közönséges Maude forrás, számos hasznos modult definiál sok kategóriá-val és operátorral, melyek közül a fontosabbak:

- Bool értékekkel kapcsolatosak: `true`, `false`, `_==_`, `_=/=_`, `if_then_else_fi`, `_and_`, `not_` ....

  ```
  if 4 - 2 == 2 then 5 of Diamonds else 6 of Spades fi
  eq M > N = not ( M <= N ) .
  ```

- Természetes számok kezelésével kapcsolatosak (támogatja a nem Peano jelölést is): `Zero`, `NzNat`, `Nat`, `_+_`, `_*_`, `_quo_`, `_rem_`, `_sd_`, relációjelek, ...
- Egész-, racionális-, valós számok kezelése
- Stringek: `_+_`, `length(_)`, `substr(_,_,_)`, `find(_,_,_)` ...
- Quoted Identifiers (`'alma`, `'józsi`), felhasználásuk

  ```
  sort Name .
  subsort Qid < Name .
  ```

Funkcionális modulok

Az első igazi Maude programunk, a Peano számábrázolás kezdeti specifikációja:

```
fmod PEANO-NAT is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor iter] .
    op _+_ : Nat Nat -> Nat .

    vars M N : Nat .
    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

Egyenleteket a `eq` kulcsszóval definiálunk. Az egyenlet két oldalán álló kifejezésnek ugyanolyan (hiba)kategóriájúnak kell lennie, bármely változónak, mely szerepel a jobb oldalon, a bal oldalon is szerepelnie kell.

A Maude az egyenleteket *egyszerűsítési szabályokként* kezeli. A Maude procedurális szemantikája az, hogy az egyenleteket tetszőleges sorrendben, balról jobbra alkalmazza mindaddig, míg nem tud tovább egyszerűsíteni. Elvárás az egyenletektől, hogy „jók” legyenek, ez a programozó felelőssége.

Funkcionális modulok - folytatás

Például az `s(s(0)) + s(0)` a következő lépésekben egyszerűsődik ki:

- `s(s(0) + s(0))` (2)
- `s(s(0 + s(0)))` (2)
- `s(s(s(0)))` (1)

Másik kezdeti példánk legyen egy faktoriális számító program, a Peano világban:

```
fmod FACT is
    protecting NAT .
    op _! : Nat -> NzNat .

    var N : Nat .
    eq 0 ! = 1 .
    eq (s N) ! = (s N) * N ! .
endfm
```

Ekkor `3 !` a szokásos módon egyszerűsödik ki.

Operátor attribútumok, jelzők

Az operátorokhoz tartozhatnak ún. **operátor attribútumok**, melyekkel többlet szemantikát, szintaktikát adhatunk egy operátorhoz. Az attribútumokat `[ ]` jelek közé írjuk:

```
op s : Nat -> Nat [ctor iter] .
op _+_ : Nat Nat -> Nat [id: 0] .
op f: Foo -> Foo [assoc comm] .
```

Egyenlet-attribútumok

Egyenleteket nemcsak az `eq` kulcsszóval definiálhatunk egy modulban, hanem operátor attribútumként is megadhatjuk őket, implicit módon. Maudeban a következő egyenlet-attribútumok lehetségesek (csak bináris operátorok esetén adhatóak meg)

- **assoc**: az operátor asszociatív
- **comm**: az operátor kommutatív
- **idem**: az operátor idempotens
- **id**, **left id**, **right id**: az operátorhoz tartozó egységelem specifikálásasa

Az egyenlet-attribútumok előnye, hogy a rendszer speciálisan (és jóval hatékonyabban) kezeli a kérdéses egyenleteket. A deklaratív szemantikát tekintve nincsen különbség aközött, hogy a `_+_` operátort `left id: 0`-nak definiáljuk, vagy felvesszük a már ismert egyenletünket

```
eq 0 + N = N .
```

Jelen esetben a procedurális szemantika **is** megegyezik, hatékonysági kérdéssé válik az attribútum használata. Ellenben például a `comm` esetében egy `X + Y = Y + X` jellegű egyenlet könnyedén okoz nemterminálódó egyszerüsítést!

Szép példa az asszociativitásra a `NatSet` kategórián értelmezett `_;_` operátor:

```
sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet .
op _;_ : NatSet NatSet -> NatSet [assoc comm id: empty] .
op _in_ : Nat NatSet -> Bool .
var N : Nat . var Ns : NatSet .

eq N ; N = N .
eq N in N ; Ns = true .
eq N in Ns = false [owise] .
```

Vegyük észre, hogy mennyire triviális a halmazbeliség eldöntésére szolgáló egyenlet:
`eq N in N ; Ns = true .`

Egy futási példa a Maude-ból

```
Maude> red 5 in 4 ; 5 ; 6 .
reduce in SET : 5 in 4 ; 5 ; 6 .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Megjegyzés: amennyiben egyenlet-attribútum(oka)t használunk átalakul a Maude procedurális szemantikája és nem egyszerű szintaktikus „egyenlőket egyenlőkkel” csere történik a meglévő egyenleteknek megfelelően.

Konstruktorok

Egy operátorról a `ctor` operátor attribútummal jelenthetjük ki, hogy az egy **konstruktor**. A konstruktorok azok, melyek egy tömör Maude kifejezés egyenlet alapú egyszerüsítése után előálló kanonikus alakban megmaradnak. Azon operátorokat, melyek eltűnnek (például két Peano szám összeadása esetén a `_+_`) **definiált függvényeknek** hívjuk.

Egyenletek, további példák

Egészítsük ki a PEANO-NAT modulunkat a szorzás műveletével:

```
fmod PEANO-NAT-MULT is
    protecting PEANO-NAT
    op _*_ : Nat Nat -> Nat .

    vars M N : Nat .
    eq N * 0 = 0 .
    eq N * s(M) = N + (N * M) .
endfm
```

Ekkor `3 * 5` a következőképpen egyszerűsödik:

```
3 * 5 = 3 + (3 * 4) = 3 + (3 + (3 * 3)) = ...
```

A listák egy lehetséges definíciója Maude-ban (érdekesség, hogy egy elem is lista)

```
fmod BASIC-LIST is
    sorts List Elt .
    subsort Elt < List .
    op nil : -> List [ctor] .
    op __ : List List -> List [ctor assoc id: nil].
endfm
```

Egy így definiált lista hossza

```
...
    op size : List -> Nat .
    var E : Elt . Var L : List .
    eq size(nil) = 0 .
    eq size(E L) = s(size(L)) .
...
```

Végül egy kártyajáték, amire a későbbiekben szükség lesz

```
fmod CARD is
    sorts Number Suit Card .
    ops 2 3 4 5 6 7 8 9 10 J Q K A : -> Number [ctor] .
    ops Clubs Diamonds Hearts Spades : -> Suit [ctor] .
    op _of_ : Number Suit -> Card [ctor] .
    op CardNum : Card -> Number .
    op CardSuit : Card -> Suit .
    var N : Number . var S : Suit .

    eq CardNum(N of S) = N .
    eq CardSuit(N of S) = S .
endfm
```

Feltételes egyenletek

Feltételes egyenleteket a `ceq` kulcsszóval definiálhatunk.

```
ceq different(N,M) = true if N =/= M .
ceq bothzero(N,M) = true if N == M /\ M == 0 .
```

Az alábbiakban bevezetjük a szokásos relációjeleket Peano számok között.

```
fmod NAT-BOOL is
    protecting NAT .
    op _<=_ : Nat Nat -> Bool .
    op _>_ : Nat Nat -> Bool .
    vars M N : Nat .

    eq 0 <= N = true .
    eq s(M) <= 0 = false .
    eq s(M) <= s(N) = M <= N .
    eq M > N = not ( M <= N ) .
    ceq N - M = 0 if M > N .
endfm
```

Megjegyzés: A könyvtári `if_then_else_fi` operátor teljesen más, mint a `ceq`-nál látott `if`.

Elemségi axiómák

Egy funkcionális modulban az egyenletek mellett a másik újdonság, az **elemségi axiómák** használatának lehetősége. Ilyet az `mb` kulcsszóval definiálhatunk, segítségével kijelenthejük egy Maude kifejezésről annak típusát (egy összefüggő komponensen belül kell, hogy legyenek).

```
vars N M : NzNat .
mb N / M : NzNat .
```

Valójában ezen axiómák képezik az alapjait a funkcionális modulok **összes** konstrukciójának. Például az alábbi operátor

```
op _+_ : Nat Nat -> Nat .
```

ekvivalens az alábbi **feltételes elemségi axiómával**:

```
cmb N + M : Nat if N : Nat and M : Nat .
```

Hasonlóképpen az alábbi alkategória definíció

```
subsort NzNat < Nat .
```

így írható át elemségi axiómává:

```
cmb N : Nat if N : NzNat .
```

Elemségi axiómák - folytatás

Bővítsük ki a kártyás modulunkat azzal, hogy minden 7-es lap „szerencsés”.

```
sort LuckyCard .
subsort LuckyCard < Card .
var C : Card .
cmb C : LuckyCard if CardNum(C) == 7 .
```

Mintaillesztés Maude-ban

Definiáljuk a póker szabályai szerinti párokat.

```
fmod CARD-PAIR is
    protecting CARD .
    sorts Pair PokerPair .
    subsort PokerPair < Pair .
    op <_;_> : Card Card -> Pair [ctor comm] .
    var N : Number .
    var P : Pair .

    cmb P : PokerPair if < N of S:Suit ; N of S':Suit > := P .
endfm
```

Mintaillesztés - folytatás

Egy rövidített futási példa, trace-szel együtt

```
Maude> red < 5 of Diamonds ; 5 of Spades > .
reduce in CARD-PAIR : < 5 of Diamonds ; 5 of Spades > .
*********** trial #1
cmb P : PokerPair if < N of S:Suit ; N of S':Suit > := P .
P --> < 5 of Diamonds ; 5 of Spades >
N --> (unbound)
S:Suit --> (unbound)
S':Suit --> (unbound)
*********** solving condition fragment
< N of S:Suit ; N of S':Suit > := P
*********** success for condition fragment
< N of S:Suit ; N of S':Suit > := P
P --> < 5 of Diamonds ; 5 of Spades >
N --> 5
S:Suit --> Diamonds
S':Suit --> Spades
*********** success #1
Pair: < 5 of Diamonds ; 5 of Spades > becomes PokerPair
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result PokerPair: < 5 of Diamonds ; 5 of Spades >
```

Funkcionális modulok - formálisan

A funkcionális modulok a Maude funkcionális *résznyelvét* alkotják. Egy modullal valójában egy *membership equational logic*-beli *elméletet*(theory) írunk le. Formálisan egy ilyen elméletet egy pár: $(\sum, E \bigcup A)$.

- $\sum$ az úgynevezett *szignatúra*, amely a típusrendszert és az operátorokat specifikálja (**kategóriák, alkategóriák, hibakategóriák, operátorok**).
- E (feltételes) **egyenletek** és **elemségi** (membership) **axiómák** halmaza
- A **operátor attribútumok** halmaza

A procedurális szemantika nem más, minthogy az egyenleteket egyszerűsítési szabályoknak tekintjük és balról jobbra egyszerűsítünk. Fontos, hogy az egyenletek „jók” legyenek, ez a programozó felelőssége.

Operátor kiértékelési stratégiák

Sok esetben nem mindegy az egyenletek végrehajtási sorrendje, mert esetleg végtelen ciklusba kerülünk. Az is elképzelhető persze, hogy az egyenletek „jók” és ilyenkor mindegy, hogy milyen sorrendben alkalmazzuk őket tömör kifejezések redukciójához, a végén úgyis a kanonikus alakhoz jutunk. A hatékonyság szempontjából azonban ekkor is **számít az egyenletek végrehajtási sorrendje**.

Ha adott egy $f(t_1, \ldots, t_n)$ kifejezés, akkor számos stratégia képzelhető el a redukció során

- meghatározzuk t_i-k redukált formáját, majd így redukáljuk f-et; ez a tipikus mohó stratégia
- meghatározunk néhány redukált t_i-t, majd megpróbáljuk f-et redukálni; példa erre az `if_then_else_fi` operátor
- megpróbáljuk redukálni f-et először, ha nem sikerül, akkor néhány argumentumát, aztán megint f-et stb. Ezek a *lusta* kiértékelési stratégiák.

A Maude teljesen a programozó kezébe adja a kiértékelési stratégia megválasztását, mely operátorról operátorra változhat. Egy n argumentumú f operátor esetén a stratégia egy lista, mely megadja a kiértékelés sorrendjét (mindig nullával végződik).

```
op foo : Nat Nat String -> Nat [strat(1 0 2 3 0)] .
```

Operátor kiértékelési stratégiák - folytatás

Az alapértelmezett kiértékelés mohó: `strat(1 2 ... n 0)` egy n argumentumú operátor esetén. A faktoriális alábbi kiszámítása végtelen ciklust ereményezne mohó kiértékelés mellett, de `strat(1 0 2 3 0)` esetén (az a `if_then_else_fi` mintája) nincsen gond.

```
eq fact(N) = if N == 0 then 1 else N * fact(N - 1) fi .
```

Megjegyzés: Az operátor kiértékelési stratégiák csak funkcionális modulokban használhatóak, pontosabban rendszer modulok esetén csak a modul egyenlet részeire van hatással.

Nagy példa lusta kiértékelésre, az Eratoszthenész-féle szita Maude megvalósítása:

```
fmod SIEVE is
    protecting INT .
    sort IntList .
    subsort Int < IntList .
    op nil : -> IntList .
    op _._ : IntList IntList -> IntList [assoc id: nil strat(0)] .
    op force : IntList IntList -> IntList [strat (1 2 0)] .
    op show_upto_ : IntList Int -> IntList .
    op filter_with_ : IntList Int -> IntList .
```

```
    op ints-from_ : Int -> IntList .
    op sieve_ : IntList -> IntList .
    op primes : -> IntList .
    vars P I E : Int .
    vars S L : IntList .

    eq force(L, S) = L . S .
    eq show nil upto I = nil .
    eq show E . S upto I = if I == 0 then nil else
       force(E, show S upto (I + - 1)) fi .
    eq filter nil with P = nil .
    eq filter I . S with P = if (I rem P) == 0 then
       filter S with P else I . filter S with P fi .
    eq ints-from I = I . ints-from (I + 1) .
    eq sieve nil = nil .
    eq sieve (I . S) = I . sieve (filter S with I) .
    eq primes = sieve ints-from 2 .
endfm
```

Példafutás:

```
Maude> red show primes upto 6 .
rewrites: 200 in 0ms cpu (0ms real) (~ rewrites/second)
result IntList: 2 . 3 . 5 . 7 . 11 . 13
```

Táblázás Maude-ban

- bármely operátort megjelölhetünk a `memo` jelzővel
- ennek jelentése, hogy a Maude tábláz minden olyan *kifejezést*, melynek legkülső operátora a `memo`-val megjelölt operátor
- naív megoldás az `n.` Fibonacci szám kiszámítására:

```
fmod fibo is
    protecting CONVERSION .
    op fib : NzNat -> NzNat .
    var N : NzNat .
    eq fib(1) = 1 .
    eq fib(2) = 2 .
    eq fib(N) = fib(N - 1) + fib (N - 2) .
endfm
```

- futási eredmények
  - N = 20 : 40ms, 33821 újraírás
  - N = 30 : 5,5s, 4160196 újraírás

- N = 40 : timeout

- táblázzuk a `fib` operátort:

```
    op fib : NzNat -> NzNat [memo] .
```

- futási eredmények
  - N = 500 : 10ms, 2491 újraírás
  - N = 10000 : 380ms, 49991 újraírás
  - N = 20000 : 1,2s, 99991 újraírás
  - N = 50000 : segmentation fault :)

- végtelen ciklusok és a tabellázás

```
...
    eq fib(N) = fib(N) .
...
```

- tabellázás nélkül végtelen ciklus, tabellázással segmentation fault, bár mindkét esetben azonos a trace

Hibakezelés

Egyenletek segítségével hibakifejezéseket „normálisokká” alakíthatunk (tehát olyanná, melynek kategóriája is van). Ennek van értelme, több okból is

- ha a `sire` operátor egy `breed(_,_)` kifejezés első argumentumát adja vissza, akkor a `sire(breed(schnauzer,penguin))` redukált alakjának `schnauzer`-nek kellene lennie
- egyfajta hibakezelést tesz lehetővé, hasonlóan mint a `try-catch` blokkok imperatív nyelvekben

```
op race : Dog Dog -> Dog . op speed : Dog -> Nat .
eq speed(pitbull) = 20 . eq speed(schnauzer) = 25 .
eq speed(penguin) = 5 . eq speed(frog) = 3 . vars N M : Dog .
eq speed(breed(N,M)) = (speed(N) + speed(M)) quo 2 .
ceq race(N,M) = N if speed(N) > speed(M) .
eq race(N,M) = M [owise] .
```

Ekkor a `race(breed(schnauzer,penguin),pitbull)` nem redukálódik, hanem [Animal] hibakategóriájú kifejezést dob vissza a Maude. Ha megváltoztatjuk a változódeklarációkat `vars N M : [Dog]`-ra, a meccs győztese a `pitbull` lesz.

Rendszer modulok

A rendszer modulok **átírási szabályokkal** (*rewrite rules*) kiegészített funkcionális modulok. Végrehajtás szempontjából a szabályok *konkurrens tranzíciókként* viselkednek, melyek tüzelhetnek, ha a szabály baloldala a rendszer állapotára illeszkedik és teljesül az őrfeltétel.

```
mod CLIMATE is
    sort Weathercondition .
    ops sunnyday rainyday : -> Weathercondition .

    rl [raincloud] : sunnyday => rainyday .
endm
```

A fenti példával ellentétben általában érdmes különválasztani a funkcionális és rendszer szintű részeket és külön-külön modulban megvalósítani azokat.

A tranzíciók állapotokat kapcsolnak össze, melyek típusainak azonos összefüggő komponensben kell lenniük.

Példa: valaki 4 cigarettacsikkből képes „gyúrni” egy új cigarettát; hány cigarettát képes elszívni, ha kezdetben 16 darab van nála?

Rendszer modulok - folytatás

```
mod CIGARETTES is
    sort State .
    op c : -> State [ctor] . op b : -> State [ctor] .
    op __ : State State -> State [ctor assoc comm] .
    rl [smoke] : c => b .
    rl [new] : b b b b => c .
endm
```

A szabályok bizonyos esetekben ütközhetnek (például, ha lenne egy `c => d` szabály), ezért fontos, hogy a számunkra megfelelő **stratégiát** használjuk. A beépített `rewrite` paramcs az alapértelmezett **stratégiát** valósítja meg:

```
Maude> rew [100] c c c c c c c c c c c c c c c c .
result State: b
```

Ahhoz, hogy megszámoljuk a cigarettákat módosítani kell a programunkon:

```
rl [smoke] : c(X) => b(X + 1) .
rl [new] : b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .
```

Rendszer modulok - nagy példa

Az alábbi példa egy, az áruházakban sokszor megtalálható plüssállat begyűjtő gép modellje.

```
mod BOX is
    protecting QID .
    sorts ToyID State .
    subsort Qid < ToyID .
    op floor : ToyID -> State [ctor] .
    op on : ToyID ToyID -> State [ctor] .
    op clear : ToyID -> State [ctor] .
    op hold : ToyID -> State [ctor] .
    op empty : -> State [ctor] .
    op i : -> State [ctor] .
    op _&_ : State State -> State [ctor assoc comm id: i] .
    vars X Y : ToyID .

    rl [pickup] : empty & clear(X) & floor(X) => hold(X) .
    rl [putdown] : hold(X) => empty & clear(X) & floor(X) .
    rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
    rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
endm
```

Átírási stratégiák

Legyen a kiinduló helyzet a következő: `empty & floor('lúd) & on('maci,'lúd) & on('labda,'maci) & clear('labda) & floor('sárkány) & clear('sárkány)`, ekkor az alapértelmezett stratégia a következő állpotba viszi a rendszert:

```
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result State: empty & floor('lúd) & clear('sárkany) & on('labda, 'maci)
& on('maci, 'lúd) & on('sárkany, 'labda)
```

Azaz a sárkányt rátettük a labdára. Sajnos, ha `rew [16]`-tal futtatunk szintén ugyanezt az eredményt kapjuk! Az alapértelmezett stratégia csak két szabályt használ (`stack` és `unstack`) és kiéhezteti a többieket.

A `frewrite` parancs (*fair rewrite*) ezt oldja meg - bár ez még mindig beépített stratégia, tehát például ne használjuk Hanoi tornyok megoldására.

Nagyon hasznos parancs a `search`, mellyek egy kezdő és végállapot közötti utat kereshetjük meg, első próbálkozásunk lehetne:

```
Maude> search empty & floor('lud) & on('maci,'lud) & on('labda,'maci)
```

```
& clear('labda) & floor('sarkany) & clear('sarkany) =>+ empty
& floor('maci) & on('lud,'maci') & on('labda,'lud)
& clear('labda) & floor('sarkany) & clear('sarkany) .
```

Melyre a válasz:

```
Solution 1 (state 70)
states: 71  rewrites: 127 in 0ms cpu (0ms real) (~ rewrites/second)
empty substitution

No more solutions.
states: 125  rewrites: 272 in 10ms cpu (10ms real) (27200 rewrites/second)
```

Ha el szeretnénk jutni egy olyan állapotba, hogy a fogókar üres és a lúd és a maci a földön van (más nem érdekel minket):

```
Maude> search empty & floor('lud) & on('maci,'lud) & on('labda,'maci)
& clear('labda) & floor('sarkany) & clear('sarkany) =>+ empty &
floor('maci) & floor('lud) & N:State .
```

Ekkor ilyen válaszokat kapunk :

```
Solution 13 (state 93)
states: 94  rewrites: 177 in 10ms cpu (30ms real) (17700 rewrites/second)
N:State --> clear('labda) & clear('lud) & on('labda, 'sarkany)
& on('sarkany, 'maci)
```

Az adott utat a `show path 93` paranccsal tudjuk megtekinteni.

Feltételes átírási szabályok

Feltételes szabályokat a `crl` kulcsszóval definiálhatunk.

```
crl [pickup] : empty & clear(X) & floor(X) => hold(X) if weight(X) < 10 .
```

Az alábbiak ekvivalensek:

```
crl [equation1] : a(X) => b(X - 1) if X > 0 .
crl [equation2] : a(X) => b(X - 1) if X > 0 == true .
crl [equation3] : a(X) => b(X - 1) if X > 0 = true .
```

```
crl [membership1] : a(X) => b(X - 1) if X :: NzNat .
crl [membership2] : a(X) => b(X - 1) if X : NzNat .
crl [pattern] : a(X) => b(X - 1) if s(N:Nat) := X .
```

Érdekesség, hogy a feltétel részben állhat egy másik szabály is. Az alábbi azt jelenti, hogy csak akkor lehet végrehajtani a szabályt `b(X)`-en, ha `a(X)` eljut valamilyen `b`-be:

```
crl [rewrite] : b(X) => c(X * 2) if a(X) => b(Y) .
```

Ennek megfelelően `a(3)`-ból eljuthatunk `c(4)`-be

```
a(3) => b(2) => c(4)
```

Ám, `a(1)`-ből nem juthatunk el semmilyen `c`-be, mert `a(0)`-ból nem tudunk semmilyen `b`-be jutni.

```
a(1) => b(0) => fail
```

Rendszer modulok - formálisan

A rendszer modulokkal egy *rewriting logic*-beli *elméletet* írunk le, mely formálisan egy négyes: $(\sum, E \bigcup A, \phi, R)$.

- $(\sum, E \bigcup A)$ a funkcionális moduloknál már ismeretett módon a szigantúrát, illetve az egyenlet és elemségi axióma (valamint az operátor attribútum) halmazokat definiálja
- ϕ a $\sum$-beli operátorok **befagyott** argumentumait adja meg
- R a (feltételes) **átírási szabályok** halmaza

A funkcionális modulok olyan speciális rendszermodulok, amik $(\sum, E \bigcup A, \phi_0, \emptyset)$ alakúak. ϕ_0 az összes operátor valamennyi argumentumát tartalmazza, azaz minden argumentumot befagyasztunk.

Full Maude rendszer

A teljes Maude rendszer az alaprendszer kibővítése, melyet Maude-ban írtak. Nevezetesen, a LOOP-MODE modult használták erre a célra, mely a felhasználóval történő interrakciót teszi lehetővé.

A LOOP-MODE használatára példa az alábbi program, ami kiírja egy felhasználó által begépelt szám kétszeresét:

```
mod NAT-LOOP is
    including LOOP-MODE .
    protecting CONVERSION .
    protecting NAT .
    op none : -> State .
    op startnat : -> System .
    eq startnat = [nil, none, nil] .

    var S : State .
    vars I O : QidList .
    var Input : Qid .
    rl [input1] : [Input I, S, O] =>
       [I, S, O qid(string(rat(string(Input),10) * 2,10))] .
endm
```

Full Maude: LOOP-MODE modul

```
Maude> loop startnat .
Maude> (12 23) .
24 46
```

A `LOOP-MODE` és így a programunk működésének alapja az alábbi konstruktor (a `System` és a `State` kutya közönséges kategóriák)

```
op [_,_,_] : QidList State QidList -> System [ctor ...]
```

A felhasználó zárójelek között megadott kifejezéseit a rendszer automatikusan `Qid`-é konvertálja és felépít belőle egy listát. Ha loop módban vagyunk, akkor implicit módon lefut egy belső szabály és kicseréli a fenti operátor alkalmazásában az első argumentumot erre. Ezután lefutnak a programozó által megírt szabályok. A tüzelés végeztével az, ami a harmadik argumentumban van kiíródik, aposztrófok nélkül.

A Full Maude nyelv nem más, mint egy Maude program, mely a `LOOP-MODE` segítségével emulálja az alap Maude-t, illetve kiegészíti azt. Két nagy újdonságot kínál fel: az objektum-orientált modulokat és a parametrizált prgramozást.

A teljes Maude rendszer Maude forrásként (fél megabájtos) áll rendelkezésünkre, induláskor be kell töltenünk:

```
Maude> load fm.maude

            Full Maude 2.0 (June 6th, 2003)

Maude>
```

Ezekután `( )` közé írt parancsokat, illetve programokat már a teljes Maude rendszer értelmezi.

Objektum-orientált modulok

Lehetőség van osztályok, alosztályok, példányok, üzenetek definiálására. Az osztályok a kategóriák, a példányok a változók, az üzenetek az operátorok „bővítései”.

Ami kimaradt...

- Szabályok fagyasztása: A `frozen` operátor attribútum hasonló módon köti meg az átírási szabályok alkalmazását, mint a funkcionális modulok esetében látott operátorkiértékelési stratégiák az egyenletek alkalmazását.
- Meta programozás Maude-ban, reflekció: sajnos talán a legfontosabb tulajdonsága a Maude nyelvnek
- model ellenőrzés
- full maude
  - parametrizált programozás Maude-ban
  - objektum-orientált modulok