Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Az Erlang nyelv párhuzamos programozást támogató nyelvi elemeinek futtatható szemantikája

Horpácsiné Kőszegi Judit
Tanársegéd, MSc

Palenik Mihály
Programtervező Informatikus MSc

Budapest, 2018

Tartalomjegyzék

1. Bevezetés

Az idők folyamán egyre több nyelv jelent meg a szoftverfejlesztés területén, mely vagy egy adott problémakörre specializálódtak, vagy általános felhasználásra készültek. Amióta megjelentek a magasabb szintű programozási nyelvek, azóta szigorú nyelvtani szabályok alapján adható meg a program, mely hibák fordítási időben kiderülnek. Mostanra kidolgozott elméleti háttérrel rendelkeznek és egy újonnan létrejövő nyelvnél nem is kérdéses a formális szintaxis kidolgozása. Oka érthető, hiszen ezek alapján a szabályok alapján parszolják a kódot és építenek szintaxis fát, ami elengedhetetlen része egy fordítónak. Nagyobb probléma viszont a szemantikai hiba, melyet futásidőben veszünk észre. Gyakorlatban az idetartozó definíciókat informálisan szöveges leírással adják meg, annak ellenére, hogy a formális szemantika definiálására is kialakult már biztos elméleti háttér. Ennek kiküszöbölésére készülnek nagy mennyiségben tesztek, melyek próbálják kisebb-nagyobb sikerrel lefedni az összes lehetséges működést, és így a szemantikai hibákra fényt deríteni.

Másik megközelítés viszont a formális szemantikadefiníció készítése lehetne. Az informális definíciók nem pontosak, félreértésekhez vezethetnek. Mivel közvetlenül nem szokás formális szemantikai szabályokat használni fordítók írásakor, nehézkes is lenne, ezért nem készítenek. Ezenfelül egy-egy vezérlési szerkezetnek nagyon bonyolult, nehezen megfogalmazható szemantikadefiníciója van, ami még egy okot ad arra a nyelv készítőknek, hogy ne foglalkozzanak formális szemantika definiálásával. Sokszor még a nyelv fejlesztői között sincs egyetértés, hogy egy bonyolult de rövid programrészletnek mi az eredménye. Szükség van formális szemantikadefinícióra, és nem csak akadémiai körökben. Nélküle nem lehetne programhelyesség bizonyítást végezni, a kérdéses esetekben az informális definíciók nem tudnak kielégítő választ adni. Sőt mi több előfordulhatnak ellentmondások, vagy olyan szemantikai szabályok, ami a többi szabály miatt nem aktiválható.

A meglévő formális szemantikadefiníciókat viszont nehézkes alkalmazni közvetlenül az előbb felsoroltakra egységesen. Ezt próbálja meg kiküszöbölni a $\mathbb{K}$ keretrendszer, amivel operációs szemantikával megadhatjuk egy nyelvnek a formális szemantikáját, és a hozzátartozó többi eszköz segítségével közvetlenül tudunk a szemantika felhasználásával verifikálni.

A felsorolt problémák és a keretrendszer lehetőségei az, ami ösztönzött diplomamunkám elkészítésére. A következő fejezetekben az Erlang nyelv párhuzamos programozását támogató résznyelvének formális szemantikadefiníciója rajzolódik ki előttünk. Sikerült nagy részét lefednem a definícióval, ám kisebb hiányosságok maradtak, mint például a monitorozás, ám a folyamatok közti kapcsolat egymás felügyelésére megvalósult, amely jó kiindulást adhat ennek befejezésére.

2. $\mathbb{K}$ keretrendszer

Diplomamunkám témája formális szemantikát definiálni az Erlang egy résznyelvéhez nem a megszokott módszerekkel, hanem a $\mathbb{K}$ keretrendszer [Fra] segítségével. Ez a keretrendszer képes az operációs szemantikai szabályok alapján egy interpretert készíteni, amivel programjaink futtathatóak, így ténylegesen megtekinthetjük az általunk definiált szemantikánk működését. Ezen felül még sok más érdekes funkcióval is rendelkezik, mint például szimbolikus programtulajdonság bizonyító és a futásidejű verifikáló. A keretrendszer fejlesztését Grigore Rosu kezdte 2003-ban [Gri16]. Jelenleg az amerikai Illinois Urbana-Champaign Egyetem és a román Alexandru Ioan Cuza Egyetem közös projektje.

Léteznek eszközök, mellyel lehet interpretert létrehozni, helyesség bizonyítani programjainkat közvetlenül a szemantika alapján, a létrehozott modellt ellenőrizni, hogy megfelel-e az adott specifikációnak, de mindezek együttesen nem jelennek meg egy keretrendszerben, emiatt különböző szemantikadefiníciókat kell létrehozni különböző célokra. Ha van egy formális nyelvdefiníciónk, akkor elvben az előbb felsoroltak megvalósíthatóak közvetlenül a nyelvdefiníciót használva, és nem kellene teljes mértékben támaszkodnunk a sok esetben ad-hoc módon implementált fordítókra. Ezt a célt tűzték ki a keretrendszer fejlesztői. A formális nyelv definiálására létrehozott módszer igyekszik kiküszöbölni az előbb ismertetett hiányosságokat, támogatást ad moduláris szemantikadefiníció létrehozásához illetve programtulajdonságok bizonyításához.

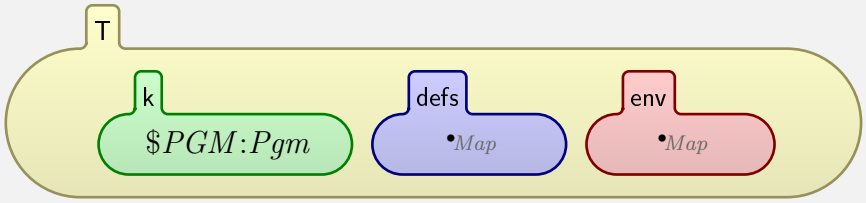
2.1. Szemantika definiálása

A keretrendszerrel teljes formális nyelvdefiníció készíthető, tehát nem csak a szemantika, hanem a szintaxis megadása is kötelező mégpedig BNF-hez hasonló formában. Ezekhez különböző attribútumokat társíthatunk. Közülük a legfontosabb a *strict*, amely a kiértékelési stratégiát határozza meg. Az ilyen fajta megadási módszer kézenfekvő, hisz szemantikaszabályok esetén már csak a kiértékelt értékekkel kell foglalkozni.

SYNTAX $Exp ::= Exp = Exp$ [strict(2)]

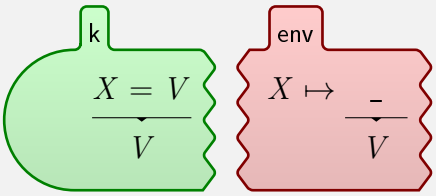
A fenti példában az értékadás operátor szintaxisa látható BNF jelöléssel, és a hozzátartozó kiértékelési stratégiával. Az operátor fajtája *Exp* és a két operandusé is ugyanaz, vagyis az értékadás mind két oldalán egy kifejezés áll, és az értékadás önmagában is egy kifejezés. A *strict* attribútum után zárójelben egy kettes van, így a meghatározott kiértékelési stratégia: csak a második operandust értékeljük ki, így az értékadó operátorhoz tartozó szemantikai szabály csak ezután alkalmazható.

CONFIGURATION:



A keretrendszerben a definiált nyelvhez tartozik egy konfiguráció, ami egymásba ágyazott cellákból áll a sorrend figyelembe vétele nélkül. Ezek tartalmazzák az összes információt, amire a programnak, illetve nekünk elemzés szempontjából szükségünk van. A konfiguráció az állapotot tartalmazza, illetve egy speciális k cellát, vagyis a számításokat, amiben a még végre nem hajtott kód található egy listában, ahol a lista minden egyes eleme egy szekvencia és a $\sim>$ szekvenciális végrehajtás jelét használja mint szeparátor. Elemei speciális K fajtájúak. A keretrendszerben minden már meglévő vagy újonnan létrehozott fajta ennek a fajtának az alfajtája. A listára bontást a keretrendszer automatikusan elvégzi. Ennek eredménye hogy a szemantikaszabályok egy egységes K fajtákat ($term$) tartalmazó számítási folyamra tudnak illeszkedni.

RULE



<k> X = V => V ...</k>
<env>... X |-> (_ => V) ...</env>

Az értékadó operátor $\mathbb{K}$ keretrendszerben kifejezett szemantikadefiníciója látható a fenti példán. Felül az olvashatóbb, prezentációra szánt változat, melyet generáltatni lehet a keretrendszerrel, illetve alul a ténylegesen írott forma látható. Két cellát a k-t és a az env-et tartalmazza. Ha a kiértékelési stratégia által meghatározott kiértékelés befejeződött, vagyis az értékadás bal oldalán egy konkrét érték található, nem pedig összetett kifejezés, akkor alkalmazható. Ezek a szabályok a redukciós szabályok, annak ellenére, ha nem is csökkentik a konfigurációt. Az átmenet a k cellában az értéket tartja meg, és ezalatt az env cellában felülírja az X-hez rendelt értéket. A _ joker karakterrel az X régi értékére illeszkedik, viszont a szabályban máshol nem akarunk hivatkozni rá, emiatt nincs nevesítve.

A szabály csak két cellát tartalmaz, azokat amik az értékadó operátor redukciós szabályában szerepet játszanak, a többit nem szükséges feltüntetni. Ennek előnye, hogy a konfiguráció könnyedén bővíthető úgy, hogy nem kell a már meglévő szabályhalmazunkon módosítani. Az alsó résznél három pont látható a k cella jobb, illetve

az *env* cella mindkét oldalán. Ezzel megadhatjuk hogy a cella elejére vagy végére akarunk illeszteni, esetleg ez számunkra lényegtelen. A szabályokhoz tartozhatnak különböző feltételek, melyek teljesülése esetén, és csak is akkor, illeszkedhetnek a szabályok a konfigurációra. A keretrendszer rendelkezik beépített típusokkal, ilyenek a teljesség igénye nélkül: halmaz, map, lista. Az *env* cella, ahogy látható is volt a konfigurációban, egy mapben tárolja a változókhoz rendelt értéket.

Térjünk vissza a *strict* attribútumhoz, hogy megértsük pontosan hogyan is működik. Példaként vegyük megint az értékadás operátort. A hozzátartozó attribútum az alábbi két szabályt generálja.

$$\textsc{rule}\ \frac{X = A;}{A \curvearrowright X = [];}$$

$$\textsc{rule}\ \frac{A \curvearrowright X = [];}{X = A;}$$

Itt megjegyezném, hogy a $\curvearrowright$ a $\sim>$ jelnek a prezentációra szánt formája. Az első szabály kiveszi az értékadó operátor kontextusából a második operandust, és berakja a folyam elejére egy új termként. A második pedig a kivett operandust visszarakja az értékadó operátor környezetébe. A kémiai absztrakt gép alapján ezeket rendre *heating* és *cooling* szabályoknak nevezzük. *strict* attribútum esetén a kiértékelési sorrend nem számít.

$$\textsc{rule}\ \frac{A1 + A2}{A1 \curvearrowright [] + A2}$$

$$\textsc{rule}\ \frac{A1 \curvearrowright [] + A2}{A1 + A2}$$

$$\textsc{rule}\ \frac{A1 + A2}{A2 \curvearrowright A1 + []}$$

$$\textsc{rule}\ \frac{A2 \curvearrowright A1 + []}{A1 + A2}$$

Itt látható szabályok abban az esetben generálódnak, ha az összeadás operátor kiértékelési stratégiája olyan, hogy mind a két operandust ki kell értékelni az összeadásra vonatkozó szemantikus szabály alkalmazása előtt. Könnyen látható, hogy előidézhetnek nem determinisztikus futást. Ugyan ezek generálódnak a *seqstrict* attribútum esetén is, de itt már számít a sorrend. Tehát az *A2*-re egészen addig nem alkalmazza a *heating*/*cooling* szabályokat, amíg az *A1*-re nem alkalmazta ezeket.

Viszont a szabályok felcserélhetőek, aminek következménye, hogy a $\mathbb{K}$-ban definiált nyelv nem lesz futtatható, mivel előfordulhat hogy nem terminál. Ennek megelőzése érdekében lett bevezetve a *KResult* fajta és a hozzá tartozó *isKResult* szemantikus függvény, amely eldönti egy termről, hogy *KResult* fajta vagy sem. A definiálandó programnyelv azon fajtáit kell a *KResult* altípusaként megjelölni, amiket értéknek tekintünk, és további redukciós szabályokat nem szeretnénk alkalmazni rá. Így a teljes generált szabályok értékadó operátor esetén az alábbiak:

$$\text{RULE}\quad \frac{X = A;}{A \curvearrowright X = [];} \qquad \text{requires } \neg_{Bool}\texttt{isKResult}\ (A)$$

$$\text{RULE}\quad \frac{A \curvearrowright X = [];}{X = A;} \qquad \text{requires } \texttt{isKResult}\ (A)$$

Először a *heating* szabályt alkalmazza, és egészen addig, amíg a kontextusból kiemelt term nem *KResult* fajtájú, nem tudja alkalmazni rá a *cooling* szabályt.

2.2. Keretrendszer további eszközei

Az előző alfejezetben láthattuk, hogyan definiálható egy formális nyelv a $\mathbb{K}$ keretrendszerrel. A kész definícióhoz a *kompile* paranccsal készíthető interpreter, és a *krun* paranccsal futtatható programokon. Alapértelmezett beállításokkal az előbb említett működés a mérvadó, viszont konfigurálható úgy, hogy az összes lehetséges kimenetelt megadja. Ez főleg konkurens programoknál fordulhat elő. Jó példa erre, mikor két szál egy globális változót akar módosítani. Az eredmény mindig függeni fog a gép órajel kiosztásától.

A keretrendszer három különböző backenddel van megvalósítva. Az elsőnek implementált verzióból lett a Maude nevű. A Java backenddel szimbolikus futtatás is lehetséges, nem úgy mint a OCaml backenddel, viszont hátránya, hogy lassabb is. A keretrendszerhez sok más eszköz is tartozik, mint például a futásidőbeli verifikáló, a statikus és dinamikus tulajdonságokat ellenőrző szimbolikus bizonyító, melynek matematikai háttere a Matching Logic [Gri15] és a Reachibility Logic [Gri12b]. A bizonyító nyelvfüggetlen ellentétben az ismert Hoare logikával. Tehát nem kell minden nyelvre külön megalkotni egy modellt a bizonyításhoz, csak a formális szemantikát kell megírnunk. A másik előnye pedig pont ez, mivel közvetlenül tudja használni a szemantikát, emiatt nem kell a szemantika és a bizonyításkor használt modell között leképezést készíteni, ami természetesen növelné a rendszer sérülékenységét is.

Most hogy már megismerkedtünk felületesen a keretrendszer által nyújtott lehetőségekkel, továbbléphetünk az Erlang nyelv világába.

3. Erlang nyelv

Az *Erlang* egy általános célra felhasználható funkcionális programozási nyelv. A Java nyelvhez hasonlóan egy szemétgyűjtővel rendelkező virtuális gépen fut a bájtkódra lefordított program. Ez a fejezet egy rövid betekintést ad az Erlang nyelvhez a reference manual alapján [Erl16]. A lent található példákhoz nem szükséges fájlba mentett, modularizált kódot használnunk, közvetlenül az Erlang Shell segítségével interaktívan is ki tudjuk értékelni kifejezéseinket. Minden kifejezés végét pont jelöli.

3.1. Adattípusok

Első körben érdemes az egyszerűbb adattípusokat áttekinteni, mielőtt megismernénk más kifejezéseket.

Két típusú szám literál létezik: *integer* és *float*.

A leggyakrabban használt literál az *atom*, gyakorlatilag egy konstans. Az aposztróf nélküli atomoknak kis betűvel kell kezdődniük és alfanumerikus karaktereket, aláhúzást és @ jelet tartalmazhatnak. Az aposztróffal rendelkező atomok Latin-1 karakterkódolású karakterből állhatnak.

```
1> erlang_20.
erlang_20
2> 'Erlang␣20'.
'Erlang␣20'
```

A *tuple* egy fix n-es, vagyis előre meghatározott számú termet tartalmazhat. A termeket kapcsos zárójelek közé, vesszővel elválasztva adjuk meg. Természetesen egymásba is ágyazhatóak.

```
1> {26, 22, {23, 23}}.
{26, 22, {23, 23}}
2> {alma, korte}.
{alma, korte}.
```

A *lista* is a tuplehöz hasonlóan egy összetett típus ellenben változó hosszúságú. A listákat szögletes zárójelekkel adjuk meg, és szintén egymásba ágyazhatóak. Kétféleképpen lehet megadni Erlang nyelvben: a termek vesszővel elválasztott formában, vagy a fej elem, maradék lista formában.

```
1> [vesszovel,elvalasztott,lista].
[vesszovel,elvalasztott,lista]
2> [fejelem | [maradek, lista]].
[fejelem,maradek,lista]
```

Egy érdekes adattípus, ami manapság már sok imperatív nyelvben is megtalálható, a *függvény objektum*. A *fun* kulcsszóval bevezetve tudunk létrehozni ilyen objektumot.

```
1> Sum = fun (X, Y) -> X + Y end.
#Fun<erl_eval.12.50752066>
2> Sum(2,3).
5
```

A példában egy függvény objektumot hoztunk létre, amely két paramétert vár X és Y, és ezeknek az összegével tér vissza. Láthatjuk is a példában, hogy a visszatérési értéke az objektumra mutató referencia. A nagy betűvel kezdődő szavak Erlangban a *változók*, később még lesz róluk szó. A *Sum* változóban tárolt függvényt zárójeles formában meg is tudjuk hívni.

A *pid* egy folyamatot azonosít három integer számmal.

```
1> spawn(module_name, function_name, Args).
<0.76.0>
```

A *spawn* függvény segítségével egy új folyamatot hozunk létre, és visszatérési értékként egy pidet ad vissza.

Még mielőtt folytatnák érdemes szót ejteni a *sztring* és a *logikai* literálokról. Erlangban mint külön adattípusok nem léteznek. A két logikai literál valójában két atom: a *true* és a *false*. A sztringek pedig listák. Idéző jelek között tudjuk megadni, de valójában az ASCII vagy unicode kódtáblában megfeleltetett számok listájaként van ábrázolva.

3.2. Mintaillesztés

Az előző részben már volt említés a változókról. A *változó* egy kifejezés. Ha egy literál kötve van a változóhoz, akkor a visszatérési értéke az a literál. Minden változóhoz csak egyszer lehetséges literált kötni, ami történhet mintaillesztés során, vagy egyszerű értékadással (ami szintén egy mintaillesztés). A változók nagy betűvel vagy aláhúzással kezdődnek, és alfanumerikus karaktereket, aláhúzást vagy @ jelet tartalmazhatnak. Az aláhúzással kezdődő változóknak speciális jelentésük van. A fordító figyelmen kívül hagyja olyan értelemben, hogy nem generál warningokat a nem használt változók miatt.

```
1> {Alma,_nemAlma,_ezsem} = {alma,nemAlma,ezSem}.
{alma,nemAlma,ezSem}
2> {Alma,_nemAlma,_nemAlma} = {alma,nemAlma,ezSem}.
** exception error: no match of right hand side value ↩
   {alma,nemAlma,ezSem}
```

A példákban mintaillesztés látható. A *minta* ugyan úgy épül fel mint egy term, ellenben tartalmazhat nem kötött változókat. *Mintaillesztés* során pedig ezekhez a változókhoz kötünk értékeket. Ha a mintaillesztés sikertelen, mint a második példában, akkor *badmatch* futásidejű hiba lép fel. A változókhoz csak egyszer köthető érték, és ez alól nem kivétel az aláhúzással kezdődőek sem, attól függetlenül, hogy a fordító figyelmen kívül hagyja.

Az anonymus változó csak egyetlen egy aláhúzásból áll. Akkor hasznos, ha kötelező változót megadnunk, de a benne lévő érték nem fontos.

```
1> {Alma,_,_} = {alma,nemAlma,ezSem}.
{alma,nemAlma,ezSem}
2> [H|_] = [33,23,12]
```

Az első példa jól mutatja, hogy gyakorlatilag figyelmen kívül hagyja az értékeket. A második példában egy lista fejelemére vagyunk kiváncsiak, amit mintaillesztéssel egyszerűen megkaphatunk.

Mintaillesztés nem csak az egyenlőség jellel lehetséges, hanem más kifejezésekben is, mint például a *receive* és a *case*, amire később láthatunk példát.

3.3. További kifejezések

Az előzőekben egyszerű kifejezésekkel ismerkedtünk meg. Ez az alfejezet pedig olyan összetettebbeket mutat be, melyeknek már meg volt a $\mathbb{K}$ keretrendszer által kifejezett szemantikadefiníciója, mielőtt neki kezdtem volna a konkurens résznyelvhez definiálni.

Az *if* kifejezés merőben eltérő az imperatív nyelvekben megszokotthoz.

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
end
```

Az *if* kifejezés ágain sorban megy végig, és ahol az első őrfeltétel (guard) igazra értékelődik ki, vagyis true atom lesz a kifejezés értéke, annak az ágnak a kifejezését értékeli ki. A visszatérési értéke pedig az ágban lévő kifejezés értéke lesz.

```
1> Animal = dog.
dog
2> if Animal == cat  -> meow;
      Animal == beef -> mooo;
      Animal == dog  -> bark;
```

```
    end.
bark
```

A példában előfordulhatna *badmatch* futásidejű hiba, ha az *Animal* változó értékét megváltoztatnánk *pig* atomra, hisz ekkor egyik ág őrfeltétele sem adna igazat.

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

A *case* esetén az *Expr* kifejezés kiértékelődik, ezután sorban végig megy az ágakon, és mintaillesztést hajt végre. Ahol a mintaillesztés sikeres, és az őrfeltétel is igazra értékelődik ki, annak az ágnak a kifejezését kiértékeli. A *case* kifejezés visszatérési értéke az ágban lévő kifejezés értéke. Szintén, ha az összes ágra sikertelen a mintaillesztés, akkor *badmatch* futásidejű hibát kapunk.

```
begin
    Expr1,
    ...,
    ExprN
end
```

A *block* kifejezések sorozata, melyek sorrendben értékelődnek ki, gyakorlatilag szekvenciát adunk meg. A visszatérési értéke az utolsó kifejezés értéke.

3.4. Függvények

A függvények deklarációja függvényklózók sorozata. Egy fejlécből áll, amely a függvény nevét, paramétereit tartalmazza és opcionálisan egy őrfeltételt, illetve egy függvénytörzsből.

```
Name(Pattern11,...,Pattern1N) [when GuardSeq1] ->
    Body1;
...;
Name(PatternK1,...,PatternKN) [when GuardSeqK] ->
    BodyK.
```

A függvény név egy atom a paraméterei pedig minták. Egy függvényt egyértelműen a modulnév a függvénynév és a hozzá tartozó aritása határozza meg.

Függvényhíváskor megpróbálja megkeresni a hivatkozott függvény deklarációját. Ha nem találja meg, akkor *undef* futásidejű hibát ad. Ha megtalálja, az első függvényklóztól kezdődően a függvény fejléceire mintaillesztést végez. Ha az illesztés

sikeres és az őrfeltétel is teljesül, ha létezik, akkor a függvénytörzset kiértékeli. Ha a mintaillesztés sikertelen, vagy sikeres illesztések esetén egy őrfeltétel sem teljesül, akkor *function_ clause* futásidejű hibát kapunk.

Ebben a fejezetben egy rövid összefoglalót láthattunk az Erlang nyelvről. Ez az áttekintés megkönnyíti a későbbi szemantikadefiníciók megértését. Természetesen ez csak egy igen apró része a nyelvnek. Arról, hogy egy valós Erlang program hogy néz ki, nem tettem említést, mivel a szemantika definiálásánál egyelőre kifejezések állnak a központban, a modulokra bontás nem. Még az Erlang konkurens részének ismertetése hátra van, de azt később a hozzájuk tartozó szemantikadefiníciók ismertetésénél részletezem.

4. Erlang nyelv konkurens résznyelvének szemantikája $\mathbb{K}$ keretrendszerben

Korábbi munkák eredményeként már létezett egy nyelvdefiníció, mely tartalmazta az alaptípusok egy részét, az ehhez tartozó operátorokat, a függvénydefiniálás és -hívás szemantikáját, mintaillesztést, néhány kifejezést mint például az *if* és a *case*, melyet részletesebben a *Meglévő szemantikadefiníció* alfejezetben fejtettem ki. A meglévő nyelvdefiníció egy korábbi (3.6-os) verzióban volt meg, és szerettem volna legújabb verzió előnyeit kihasználni a diplomamunka kidolgozásakor.

4.1. Meglévő szemantikadefiníció

A meglévő szemantikadefiníciót modularizáltam, hogy a konkurens elemekkel együtt jól áttekinthető és kezelhető legyen.

tokens.k Ebben a fájlban van az Erlang specifikus literáloknak és a változóknak a szintaxisa, mely két modult tartalmaz.

```
MODULE TOKENS-PARSING
    SYNTAX  UnquotedAtom ::= r"[a-z][_a-zA-Z0-9@]*"  [token, autoReject, notInRules]
    SYNTAX  Atom ::= UnquotedAtom
                   | Bool
    SYNTAX  Joker
    SYNTAX  Variable ::= Joker
END MODULE
```

A *TOKENS-PARSING*, ahogy látható felül, az *UnquotedAtom*, *Atom*, *Joker* és a *Variable* fajtákat tartalmazza. A definíciók maguktól értetendőek. A *Bool* és az *UnquotedAtom* alfajtája az *Atomnak*, viszont az aposztrófok közé írt atom nem, mivel egyelőre nincs definiálva. A *Joker* és a *Variable* definíciója ebben a modulban nem látható. Ennek az az oka, hogy a $\mathbb{K}$ keretrendszer rendelkezik egy *#KVariable* fajtával, amelynek a definíciójában megadott reguláris kifejezés által meghatározott lehetséges karakterláncok halmaza és a hasonló módon *Variable* által meghatározott halmaz metszete nem üres, emiatt ha itt definiálnánk, parszolási hibát kapnánk.

```
MODULE TOKENS-SYNTAX
  IMPORTS TOKENS-PARSING
    SYNTAX  Variable ::= r"[A-Z][_a-zA-Z0-9@]*"  [token, autoReject]
```

```
    SYNTAX  Joker ::= _ [token]
END MODULE
```

A TOKENS-SYNTAX modulban megadva a definíciókat ezt a hibát elkerülhetjük. Ilyenkor a parszolás után kapott fán lévő levelek *#KVariable* címkéit lecseréli a saját *Variable* fajtával.

exp-shared.k Az *EXP-SHARED* modul az *EXP* modulból – később kerül ismertetésre – leválasztott egység.

```
MODULE EXP-SHARED
    SYNTAX  Exp
    SYNTAX  Exps ::= List{Exp,","} [strict]
    SYNTAX  Match0 ::= Exp -> Exp [match0]
    SYNTAX  Match ::= List{Match0,";"} [match0]
   ...
END MODULE
```

Az *Exp* – a kifejezések fajtája – deklarálása itt található. Szinte az összes modul használja az *Exp* és az *Exps* fajtákat, viszont vannak olyanok, ahol szükségtelen az egész *Exp*-hez tartozó szintaxist importálni. Ha a modulokban újra lenne deklarálva, az névütközéshez vezetne. Az *EXP-SHARED* modul ezt hivatott elkerülni.

Ezen felül még definiálva lett az *Exps* fajta, ami vesszővel elválasztott *Exp*-ek sorozata a *strict* attribútummal, mely a kötelező kiértékelést jelöli, sorrendet figyelmen kívül hagyva. A *Match0* a *case* és az *if* kifejezések által is használt részkifejezések, amely a mintát és az utána lévő kifejezést tartalmazza, a *Match* pedig ezek pontosvesszővel elválasztott sorozata. Fontos megjegyezni, hogy ezek a részkifejezések kiértékelése sorban történik. Vagyis mindig csak a mintaillesztés során a nyíl előtti rész, majd egyezés esetén pedig annak az "ágnak" a kifejezése értékelődik ki.

operators.k Az *OPERATORS-PARSING* modulban bővítjük a kifejezések szintaxisát aritmetikai, összehasonlító és logikai operátorokkal.

```
MODULE OPERATORS-PARSING
  IMPORTS TOKENS-PARSING
   SYNTAX  Exp ::= not Exp [strict, arith]
                 | Exp * Exp [strict, arith]
                 | Exp div Exp [strict, arith]
                 | Exp rem Exp [strict, arith]
                 | Exp + Exp [strict, arith]
                 | Exp - Exp [strict, arith]
```

```
                | Exp < Exp [strict, arith]
                | Exp =< Exp [strict, arith]
                | Exp > Exp [strict, arith]
                | Exp >= Exp [strict, arith]
                | Exp == Exp [strict, arith]
                | Exp /= Exp [strict, arith]
                | Exp andalso Exp [strict(1), arith]
                | Exp orelse Exp [strict(1), arith]
END MODULE
```

Az *OPERATORS* modulban pedig az ezekhez tartozó szemantikadefiníciók találhatóak. Mivel a szintaxis esetén megadtuk minden egyes operátornál a *strict* attribútumot, ezért a szabály alkalmazása várat magára egészen addig, amíg az operandusokat ki nem értékelte.

```
RULE  I1 div I2          requires I2 =/=Int 0
      ---------
      I1 ÷Int I2
```

Ahogy a példában látható, az aritmetikai operátorok szemantikájának definiálása esetén nagy segítséget jelentenek a $\mathbb{K}$ keretrendszer beépített operátorai.

tuple.k Ebben a fájlban a *tuple* típussal kapcsolatos szintaxis és szemantika található. A *TUPLE-PARSING* modul bővíti az *Exp* fajtát a *tuple* szintaxisával.

```
MODULE TUPLE-PARSING
  IMPORTS EXP-SHARED
   SYNTAX  Exp ::= { Exps } [strict, tuple]
END MODULE
```

A *TUPLE* modul az *OPERATORS* modul kibővítése *tuple* specifikus összehasonlító operátorokkal.

```
RULE  {(X:Value , Xs:Values)} < {(Y:Value , Ys:Values)}
      ---------------------------------------------------
      (X < Y) orelse (X == Y andalso {Xs} < {Ys})
        count (Xs) ==Int count (Ys) ∧Bool
requires count (Xs) >Int 0 ∧Bool                    [structural]
         count (Ys) >Int 0
```

Egy strukturális átalakítás látható, ami jó példa arra, hogyan lehet felhasználni, a már meglévő szemantikadefinícióinkat újabbak definiálására.

list.k Az *ERL-LIST-PARSING* modulban az Erlangban használatos lista két fajta szintaxisa található.

```
MODULE ERL-LIST-PARSING
  IMPORTS EXP-SHARED
   SYNTAX  Exp ::= [ Exps ] [strict, list]
                 | [ Exps | Exp ] [strict, list]
  ...
END MODULE
```

Az *ERL-LIST* modul csak átalakítási szabályokat tartalmaz, melyet a *macro* attribútum jelöl. Ezeket a keretrendszer a legelső számítási lépés előtt elvégzi.

```
MODULE ERL-LIST
  IMPORTS ERL-LIST-PARSING
  IMPORTS OPERATORS
  IMPORTS CONFIG
  RULE  [•Exps | —]        [macro]
        ---------
        "badlist"

  RULE  [X , Y]            [macro]
        ---------
        [X | [Y]]

  RULE  [X , Xs | Y]       requires Xs ≠K •Exps      [macro]
        ---------
        [X | [Xs | Y]]
END MODULE
```

errors.k Az *ERRORS* modul a futásidejű hibák fajtáit tartalmazza.

```
MODULE ERRORS
  IMPORTS EXP-SHARED
   SYNTAX  Error ::= $error_badmatch
                   | $error_badarg
                   | $error_noproc
   SYNTAX  Exp ::= Error
END MODULE
```

value.k Előző modulokban látható, hogy szintaxis esetén egyes fajtákhoz *strict* attribútum van rendelve. Ennek a következménye, hogy a szemantikai szabályok esetén csakis akkor illeszkedik egy minta, ha a keretrendszer a szabály előfeltételében szereplő összes fajtát a $\mathbb{K}$ specifikus *KResult* fajtára átalakította. A *VALUE* modulban ezeket gyűjtöttük össze a *Value* fajtába.

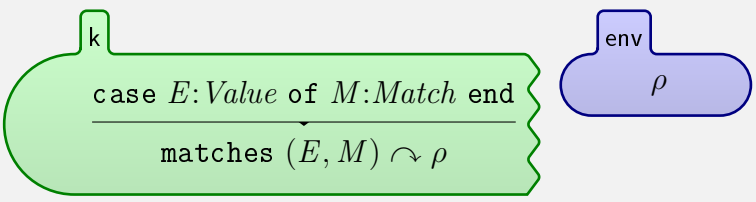
```
MODULE VALUE
  IMPORTS EXP-SHARED
  IMPORTS TOKENS-PARSING
  IMPORTS ERRORS
   SYNTAX  BasicValue ::= Atom [value]
                        | Int [value]
                        | Bool [value]
   SYNTAX  ListValue ::= [ Values ]
                       | [ Values | Value ]
   SYNTAX  Value ::= BasicValue
                   | ListValue
                   | { Values }
                   | Error
   SYNTAX  Values ::= List{Value,","}
   SYNTAX  Exp ::= Value
END MODULE
```

exp.k Az *EXP-PARSING* modul tartalmazza azon kifejezések szintaxisát, melyekhez lett szemantika definiálva.

```
MODULE EXP-PARSING
  IMPORTS EXP-SHARED
  IMPORTS TOKENS-PARSING
   SYNTAX  Exp ::= Atom
                 | Int
                 | Variable
                 | Atom ( Exps ) [strict(2), funcall]
                 | case Exp of Match end [strict(1), case]
                 | if Match end [if]
                 | begin Exps end [block]
                 | Exp = Exp [strict(2), matchexpr]
                 | ( Exp ) [bracket]
END MODULE
```

Az *EXP* modul pedig az ezekhez tartozó szemantikadefiníciókat tartalmazza.

```
RULE
```



A kiemelt példában érdemes megfigyelni, ahogy a *case* szabálya környezetet vált. A régit, amit a *Rho* nevű változó jelöl, átemeli a számítási folyamba. Mikor a *case* kifejezés visszatérési értékét megkapjuk, akkor ugyan ebből a folyamból visszaolvassa a régi környezetet.

erl-parsing.k Az *ERL-PARSING* modul egyesíti az összes *-PARSING* végű modult.

```
MODULE ERL-PARSING
  IMPORTS TOKENS-PARSING
  IMPORTS EXP-PARSING
  IMPORTS OPERATORS-PARSING
  IMPORTS ERL-LIST-PARSING
  IMPORTS TUPLE-PARSING
  IMPORTS CONCURRENT-PARSING
   SYNTAX  FunCl0 ::= Atom(Exps) -> Exp; [funcl0]
   SYNTAX  FunCl1 ::= Atom(Exps) -> Exp. [funcl1]
   SYNTAX  FunCl ::= FunCl0
                   | FunCl1
   SYNTAX  FunDefs ::= FunCl
                     | FunDefs FunDefs [right]
  ...
   SYNTAX  Pgm ::= FunDefs --- Exp.
END MODULE
```

Ezen felül meghatározza a különböző kifejezések közötti prioritásokat is. Jelenleg az Erlang modul rendszer szintaxisa még nincs kidolgozva. Egy fájl felépítése két nagy részből áll: a függvénydefiníciós rész és a program rész, amit $---$ jel választ el egymástól.

erl.k Az *ERL* modul három dologért felelős.

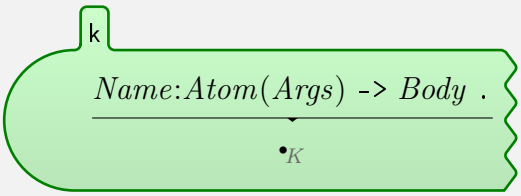
```
SYNTAX  KResult ::= Value
                  | Values
                  | Error
```

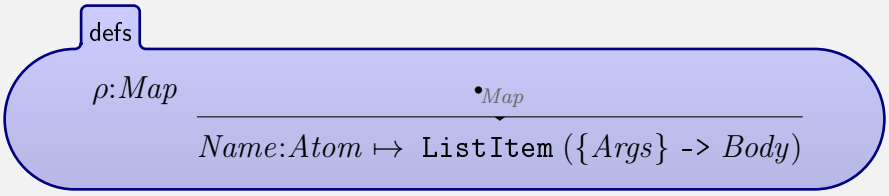
Először is a *Value* alfajtája lesz a *KResult* fajtának.

```
RULE  F:FunDefs --- E:Exp .        [structural]
      ------------------------
      F:FunDefs ↷ E:Exp
```

RULE $\dfrac{F1{:}FunCl \quad F2{:}FunDefs}{F1{:}FunCl \curvearrowright F2{:}FunDefs}$ [structural]

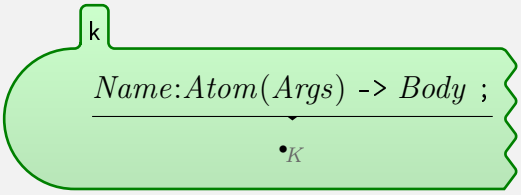
RULE

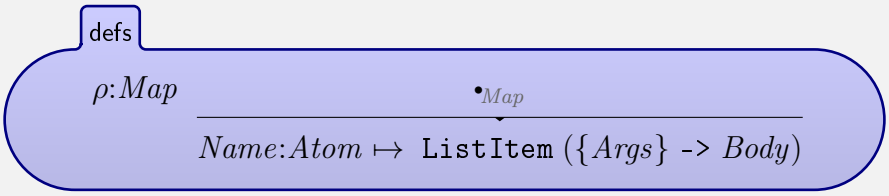




requires $\neg_{Bool} Name{:}Atom$ `in keys` (ρ) [structural]

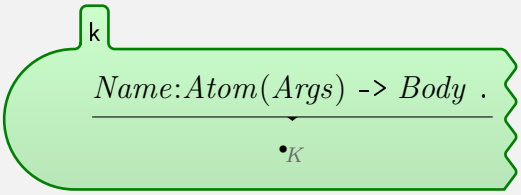
RULE





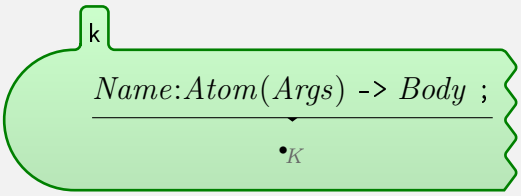
requires $\neg_{Bool} Name{:}Atom$ `in keys` (ρ) [structural]

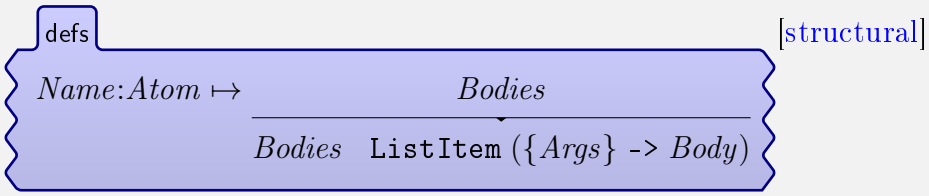
RULE



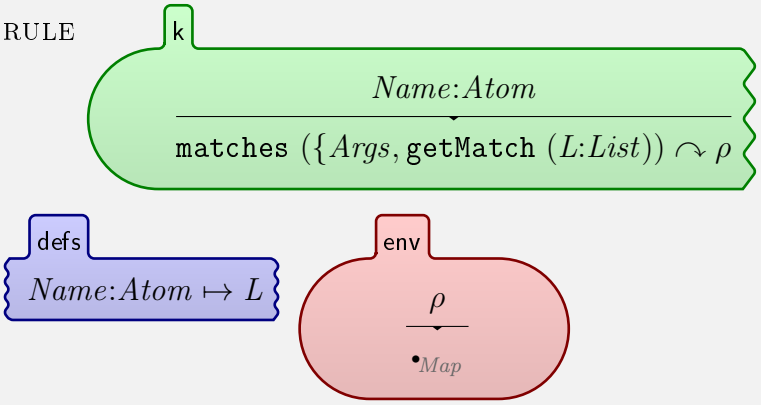
[structural]

RULE



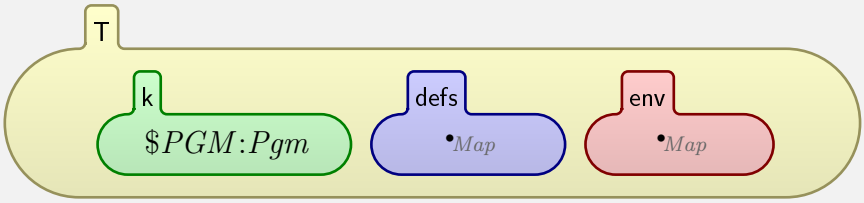


Másodszor a program fájl feldolgozását definiálja. A függvénydefiníciós részt a későbbi függvény hívások miatt a *defs* állapotba elmenti.



Harmadszorra pedig az általunk definiált függvények hívásának szemantikáját definiálja. Itt a megszokott módon környezetet vált, és a *defs* állapotban megkeresi a hozzá tartozó függvénydefiníciót.

config.k Ebben a fájlban a kezdő konfiguráció található.



A fenti kezdő konfiguráció ezen diplomamunka előtti állapotot mutatja. A k cella tartalmazza az adott konfigurációban a hátralévő programot. Az állapot jelenleg két parciális függvényből áll.

$$defs : Atom \to Lista \quad \text{ahol} \quad Lista \subseteq \{f | f : Exps \to Exp\}$$

Ahogy feljebb olvasható a *defs* tartalmazza a függvénydefiníciókat: a függvény nevéhez hozzárendeli a lehetséges függvénytörzseket. Itt a *Lista* alatt a $\mathbb{K}$ keretrend-

szer beépített listája értendő.

$$env : Variable \rightarrow Value$$

Az *env* függvény pedig a változó nevéhez hozzárendeli a változóhoz kötött értéket.

Ebben az alfejezetben egy rövid és tömör áttekintést kaphattunk a már meglévő szemantikadefiníciókról. Mielőtt tovább haladnánk, hogy megismerjük a konkurens résznyelvhez tartozó szemantikadefiníciókat, érdemes egy rövid betekintést nyerni az Erlang nyelv folyamatainak világába.

4.2. Erlang folyamatok

Az Erlang nyelv fő erőssége a konkurens és az elosztott programozásban rejlik. Konkurencia alatt a különböző szálak egy idejű kezelését értjük. Ez természetesen lehet valós párhuzamos megvalósítású több magos gépek esetén, illetve főként egy magos környezetben a szálak folyamatos cserélgetésével, így a párhuzamosság látszatát kapjuk. Ilyen szálakat Erlang nyelven könnyedén létrehozhatunk és eszközt is ad az egymás közötti kommunikációra. Erlangban ezeket a szálakat folyamatoknak hívjuk. Ez logikus megnevezés mivel a folyamatok egymással nem osztanak meg közvetlenül adatot. Az információcsere folyamatok között üzenetküldéssel történik.

Egy ilyen folyamatot a *spawn(Modul, Exportált_Függvény, ArgumentLista)* hívással tudunk létrehozni.

```
-module(representSpawn).

-export([start/0, add5/]).

add5(X) -> io:format("~p~n", [X + 5]);

start() ->
    spawn(representSpawn, add5, [3]),
    spawn(representSpawn, add5, [4]),
    spawn(representSpawn, add5, [5]),.
```

A *representSpawn* modul egy egyszerű példát mutat a spawn használatára. A *start* függvénye három darab folyamatot indít el egymás után, ami függetlenül fog futni a létrehozó folyamattól. Ha meghívjuk a start függvényt az egyik lehetséges eredmény az alábbi lehet.

```
1> representSpawn:start().
8
10
```

```
<0.45.0>
9
```

Látható hogy a kiírás sorrendje ebben az esetben 8, 10, 9. Ezen a sorozaton jól látható, hogy a folyamatok egymástól függetlenül futnak, mivel a processzor úgy ütemezett, hogy a 4-es argumentummal meghívott folyamat írja ki az eredményét a legkésőbb. A zárójelben ponttal elválasztott számok a folyamatazonosító. Minden folyamathoz tartozik egy ilyen egyedi azonosító, ami a *spawn* függvény visszatérési értéke.

Mivel a folyamatok nem osztják meg közvetlenül saját adataikat, az egymással való kommunikáció üzenetküldéssel folyik. Ezt legegyszerűbben a klasszikus ping-pong példán lehet bemutatni [Erl16].

```
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping␣finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping␣received␣pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong␣finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong␣received␣ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

A példában két folyamat kommunikál egymással. Az üzenet, akár egy ping pong labda, halad az egyik folyamattól a másik folyamatig. A *start* függvény hozza létre a két folyamatot, amelyek kommunikálnak egymással. Az első folyamat, amit létrehoz, a *pong* függvényt hívja meg, és eltárolja a folyamatazonosítóját egy változóba.

A *pong* függvény egy *receive* kifejezést tartalmaz. Ez a szerkezet felelős az üzenetek fogadásáért. Minden folyamathoz tartozik egy postaláda, amibe az üzenetek érkeznek. Ezeken a *receive* minden ágára, kezdve az elsővel, mintaillesztést végez a beérkezésük sorrendjében, amíg egyezést nem talál. Ha sikeres volt a mintaillesztés, akkor azt az üzenetet eldobja a postaládából, és az adott ághoz tartozó kifejezést kiértékeli. Ha nem volt egyezés, vár amíg újabb üzenet nem jön, és újra a legrégebbi üzenettől kezdi el a mintaillesztést. Jelen esetben nagy valószínűséggel a *pong* folyamat postaládája üres, ezért üzenetfogadására vár.

A *start* létrehozza a második folyamatot, ami a *ping* függvényt hívja meg két paraméterrel, egy számmal és egy folyamatazonosítóval. Az első paraméter megadja, hogy az üzenetküldés hányszor menjen végbe oda-vissza, a második paraméter pedig egy folyamatazonosító, amely folyamattal kommunikálni fog. A *ping* függvényben az első kifejezés a ! operátor. A két aritású operátorban az első operandusban megadott folyamatazonosító által meghatározott folyamatnak elküld egy a második operandusban megadott üzenetet. Itt az üzenet egy *tuple*, ami egy atomot és a *self* függvényt tartalmazza. A *self* visszaadja az adott folyamat folyamatazonosítóját.

Ezalatt a másik várakozó folyamat üzenetet kapott a *ping* függvényt kiértékelő folyamattól. A második ágra illeszkedik az üzenet így kiírja hogy megérkezett, és nyomban válaszol is rá a ! operátorral. A válasz már csak egy atom, mivel a *ping* függvényt kiértékelő folyamat paraméterként megkapta a folyamatazonosítót. Ezután rekurzívan meghívja önmagát. Ekkor a *ping* vagy épp most kezdi el a *receive* kiértékelését, vagy már el is végzett egy mintaillesztést, és egy új üzenetre vár. A kapott üzenet illeszkedik a *pong* atomra így jelzi, hogy az üzenet megérkezett, és a függvény szintén rekurzívan meghívja önmagát, ám az üzenetküldés számát meghatározó paraméter értékét csökkenti eggyel. Ez egészen addig megy így, amíg ez az érték nulla nem lesz. Ekkor a *ping* másik paraméterezésére fognak illeszkedni a függvényhívásban megadott paraméterek. A *finished* atomot tartalmazó üzenetet küldi el, ami jelzi a *pongnak*, hogy az üzenetküldések folyamatának vége. Mind a két függvény kiírja, hogy befejeződött az üzenetküldés, és már nincs rekurzív hívás, így a függvények kiértékelődtek, a két folyamat terminál.

Lefuttatva a példa programot, feltételezve hogy a konzolra való kiíratást egyből végrehajtja, az alábbi eredményt kapjuk.

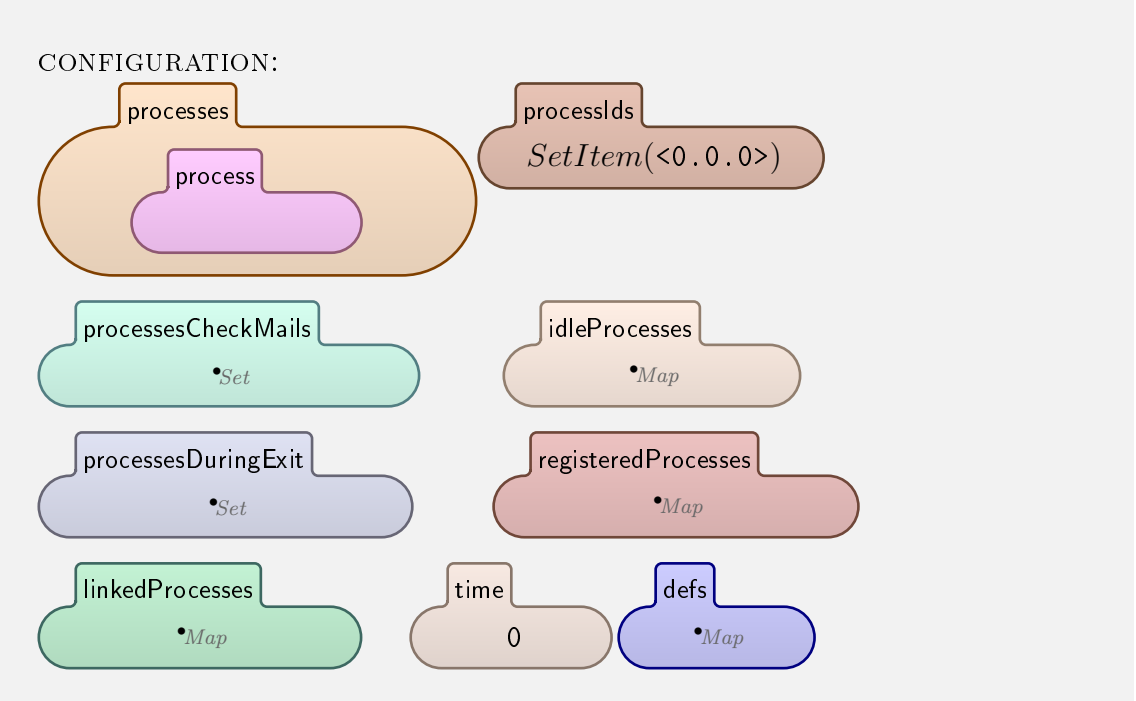
```
1> c(tut15).
{ok,tut15}
2> tut15: start().
```

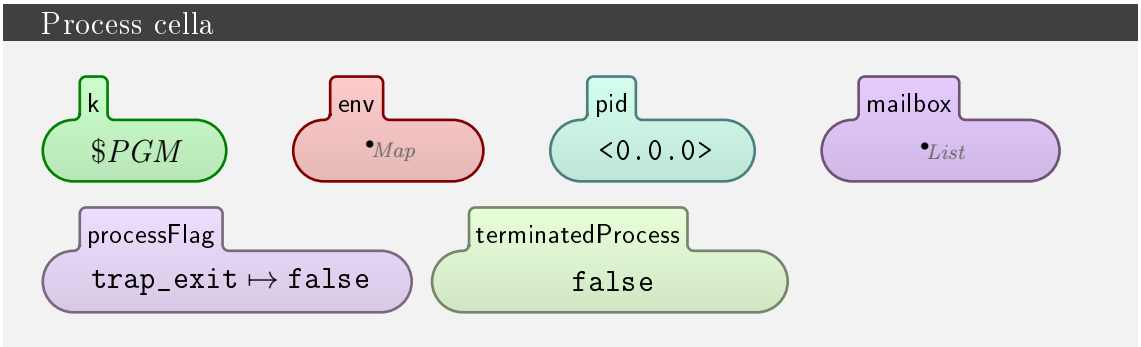
```
<0.36.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
```

Ezzel az áttekintővel egy kép alakult ki a folyamatok működéséről és kommunikációjáról, ami minimálisan szükséges a konkurens szemantikadefiníciók könnyebb megértéséhez. A diplomamunka nem tér ki a több gépet tartalmazó elosztott programozásra.

4.3. Konkurens résznyelv szemantikája

Szemantikadefiniálás gyakorlatilag a manuál értelmezéséből áll, amiből aztán megfelelő módszerrel, technikával formális szemantikadefiníciót készítünk. Ebben az alfejezetben megvizsgáljuk, hogy a nyelvi elemekhez, amihez formális szemantikát akarunk definiálni, milyen informális leírás tartozik, és ezek alapján hogyan jutunk el a formális definícióig. A konfiguráció is bővült.





A kezdő konfiguráció kiegészült pár állapottal:

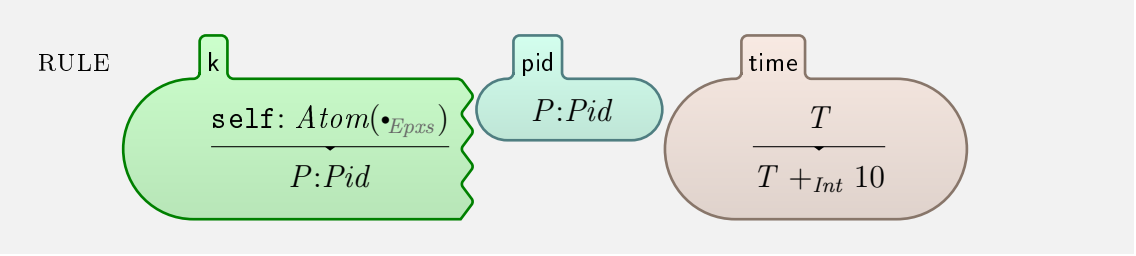
- processes: a folyamatok csoportját tároló cella
  - process: folyamatok állapota
    - k: a programot szekvenciákra bontott listaként tároló cella
    - env: a változókhoz hozzárendelt értékek az adott környezetben
    - pid: folyamatazonosító
    - mailbox: a folyamat postaládája
    - processFlag: folyamathoz tartozó flagek és értékei
    - terminatedProcess: jelzi hogy a folyamat terminált-e
- processIds: a futó folyamatok azonosítói
- processesCheckMails: folyamatok azonosítói, melyek mintaillesztést végeznek a postaládában lévő üzeneteikre
- idleProcesses: üzenetre várakozó folyamatok azonosítói, és a hozzájuk tartozó lejárati idő
- processesDuringExit: az éppen terminálló folyamatok azonosítói
- registeredProcesses: a regisztrált folyamatok azonosítói és a hozzátartozó nevek
- linkedProcesses: egymáshoz kapcsolt folyamatok
- time: az eltelt idő
- defs: függvénydefiníciók

A cellák bővebb leírása és tartalma azoknál a definícióknál lesz kifejtve, ahol először megjelennek. A résznyelv egész szemantikadefiníciója megtekinthető a diplomamunka mellékletében.

4.3.1. Folyamat létrehozása

A folyamatok létrehozásáról az előző alfejezetben már tettem említést, ami a spawn függvénnyel lehetséges. A konfigurációban a *processes* állapot tartalmazza a folyamatokat, ami gyakorlatilag egy multihalmaz (a multihalmaz jellege sose lesz kihasználva, mivel a folyamatazonosító mindig egyedi minden folyamatnak, emiatt nem létezhet két azonos elem a multihalmazban). Minden sikeres spawn híváskor egy új *process* elem kerül be a konfigurációba. Ez a process tartalmazza a már említett k cellát és az env állapotot.

A spawn függvénynek négyféle paraméterezése létezik. A három paraméterből álló szemantikadefiníciója lett megvalósítva. Az első paramétere a modul neve, amiben a függvény található. Mivel a modul rendszer szintaxisa és szemantikája még nem lett definiálva $\mathbb{K}$ keretrendszerben, emiatt az itt lévő atomot mindig figyelmen kívül hagyjuk. Második paramétere a függvény neve, harmadik pedig egy lista, ami a második paraméterben meghatározott függvény aktuális paramétereit tartalmazza. A konfiguráció átmenet létrehoz egy új process elemet, vagyis egy új folyamatot, a kezdőkonfiguráció szerinti állapot szerint. Egyedül a k cellát és a *pid* állapotot módosítja. A k cellába a függvény hívás kerül a megadott paraméterekkel, amit a *GetCommaSeparatedValuesFromList* szemantikus függvény a megfelelő alakra hoz, vagyis az átadott listából egy Values fajtára alakít. A pid állapot pedig egy egyedileg generált folyamatazonosítót fog tartalmazni, ami szintén bekerül *processIds* állapot halmazába. A processIds a futó folyamatok folyamatazonosítóinak a listája. Egyelőre csak a helyes működés van definiálva, a függvény által dobott futásidejű hibák nincsenek. Ilyenek például, ha rossz függvény nevet, vagy aritású függvényt adunk meg.



A self függvény szemantikadefiníciója egy szabályból áll, ami fent látható. A self visszatérési értéke az éppen futó folyamat folyamatazonosítója. A szabály a saját process cellájában lévő pid állapotra illeszkedik, és ezt az egyedi azonosítót tartalmazza a k cella az átmenet után, amit a folyamat létrehozásakor generáltunk.

4.3.2. Folyamatok azonosítása

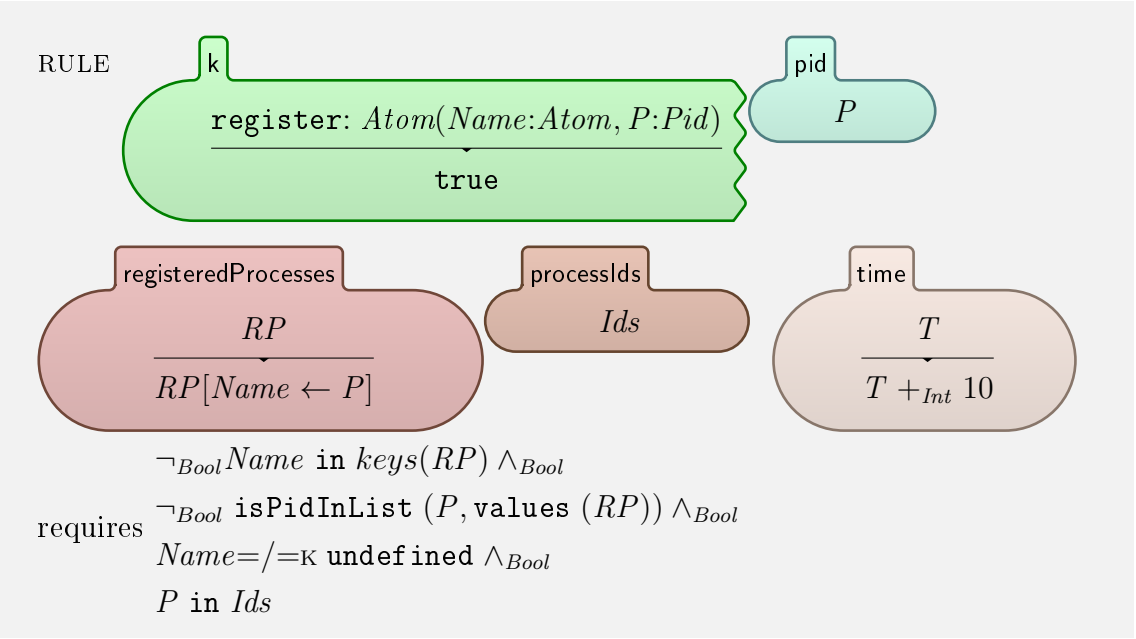
A folyamatokra a már előbb bemutatott folyamatazonosítóval is tudunk hivatkozni, ám nevesíthetjük is azokat, a két paraméteres *register* függvénnyel. Az első paramétere az általunk választott név, ami egy atom, a második pedig a folyamatazonosító,

amit ehhez a névhez szeretnénk társítani. Visszatérési értéke mindig true. *badarg* futásidejű hibát okozhat az alábbi esetekben:

- ha az első argumentumban megadott névhez már lett folyamat rendelve
- ha a második paraméterben megadott folyamatazonosító nem létezik
- ha a második paraméterben megadott folyamatazonosítóhoz már rendeltünk nevet
- ha az első paraméter az *undefined* atom

A függvénnyel több gépet tartalmazó környezetben folyamatazonosító helyett portot is átadhatunk paraméternek. Ahogy már említettem, a szemantikadefiníció nem tér ki az ilyen esetekre.

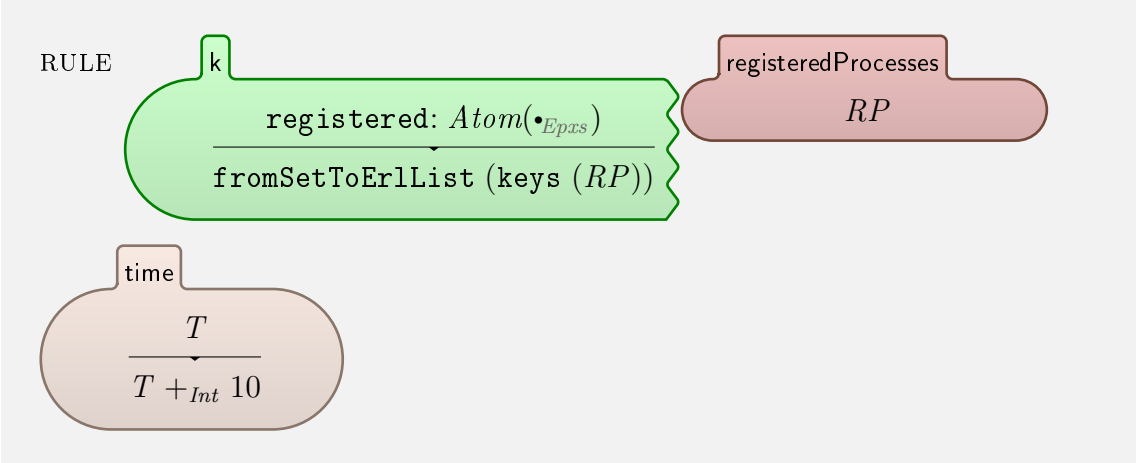
Ehhez a szemantikadefinícióhoz három szabály is tartozik.



Két szabály a sikeres név-folyamatazonosító hozzárendelése esetén, egy pedig a badarg futásidejű hiba esetén való átmenet. A fenti szabály a sikeres hozzárendelés nem saját folyamatazonosító esetén. Látható, hogy egy másik folyamat pid cellájára illeszkedik, amit a P változóba ment el, és a hozzátársított névvel ($Name$) a *registeredProcesses* állapot map-ét bővíti az átmenet. Ez az állapot név-folyamatazonosító párokat tartalmaz, ami a nevekhez való társításokat tárolja. Az illeszkedés esetén való feltételt érdemes részletesebben megvizsgálni. Ez gyakorlatilag az előbb listaként felsorolt állítások tagadásai. Az RP-ben lévő párok nem tartalmazhatják külön a regisztrált nevet, illetve a folyamatazonosítót, nem lehet a név undefined atom, és a folyamatazonosítónak benne kell lennie az Ids halmazban, ezzel kötjük ki, hogy csak futó folyamatokhoz tudunk nevet társítani.

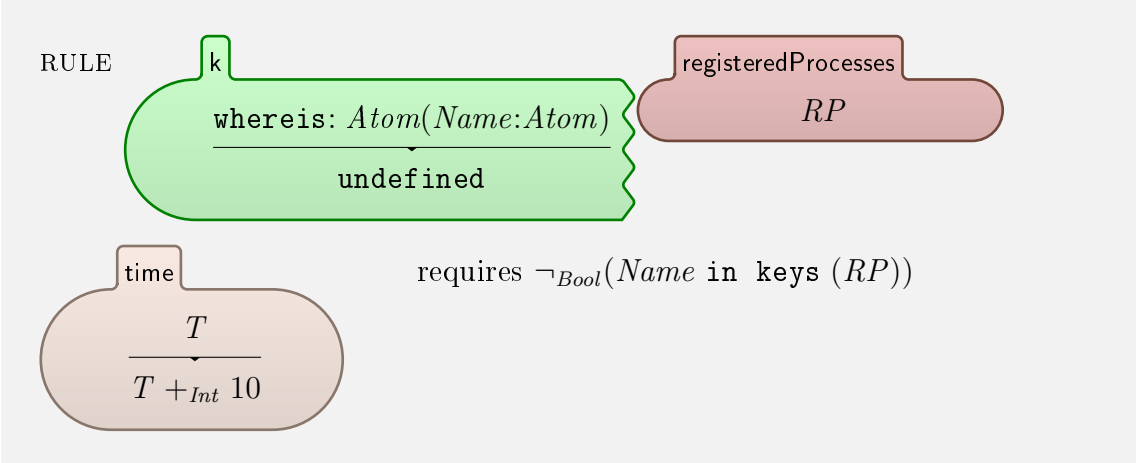
A másik sikeres szabály hasonló az előzőhöz, azzal a kivétellel, hogy itt a folyamat a saját azonosítójához akar nevet társítani, így nem egy process cella pid állapotára illeszkedik, hanem a sajátjáéra. A badarg futásidejű hibát kiváltó szabály esetén a k cellában lévő átmenet ezt a hibát adja, a feltételek pedig az előző szabályokénak a negáltja.

Az így regisztrált nevek lekérhetőek a *registered* paraméternélküli függvénnyel. A neveket egy listában adja vissza.



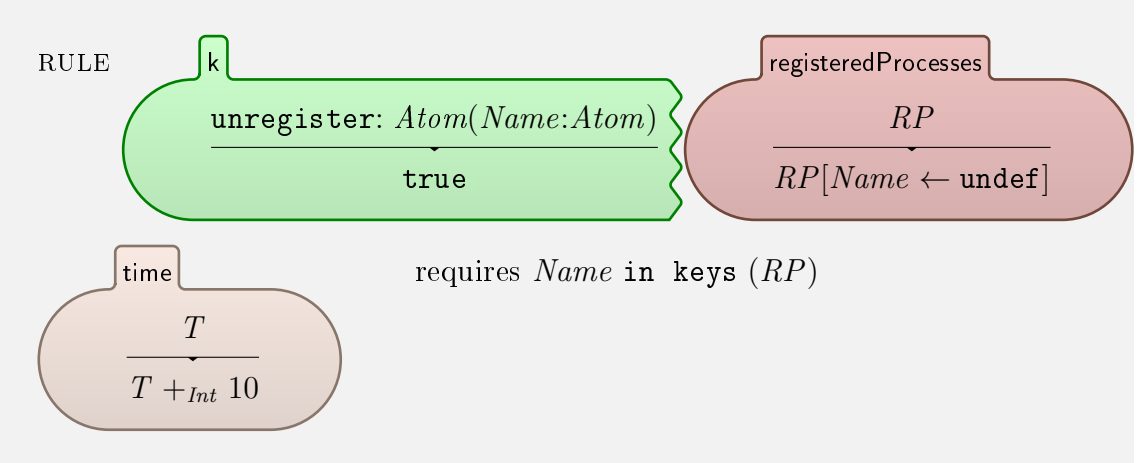
A fent látható szabály a *registeredProcesses* állapotra illeszkedik, és az RP változóba olvassa be a név-folyamatazonosító párokat. A *fromSetToErlList* szemantikus függvény a paraméterként átadott neveket tartalmazó halmazból Erlang listát képez, ami az átmenet eredménye lesz a k cellában.

A *whereis* függvény megadja a paraméterben átadott atomhoz tartozó folyamatazonosítót. Ha még nem volt hozzárendelés az undefined atommal tér vissza.



A szemantikadefiníció két szabályból áll. A fenti írja le a működést abban az esetben, ha az atomhoz nincs hozzárendelve folyamatazonosító. Az illeszkedéssel az RP változóba beolvassa a regisztrált név-folyamatazonosító párokat, és ha a nevekből alkotott halmaz nem tartalmazza a *Name* változóban lévő atomot, akkor a k cellában az átmenet az undefined atomot adja. A másik szabály találat esetén visszaadja a Name-hez rendelt folyamatazonosítót a mapből.

A register függvény segítségével létrehozott név-folyamatazonosító társítást törölhetjük az *unregister* segítségével. Visszatérési értéke mindig *true*. Ha a paraméterében átadott atomhoz nincs folyamatazonosító rendelve, akkor *badarg* futásidejű hibát okoz.

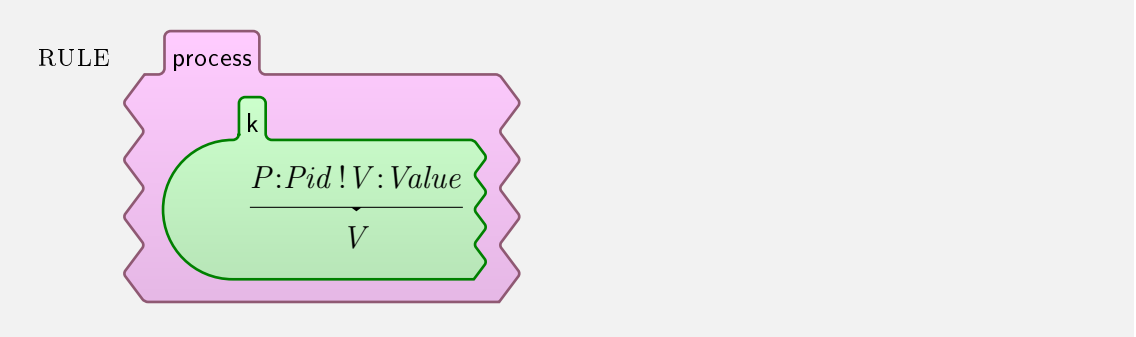


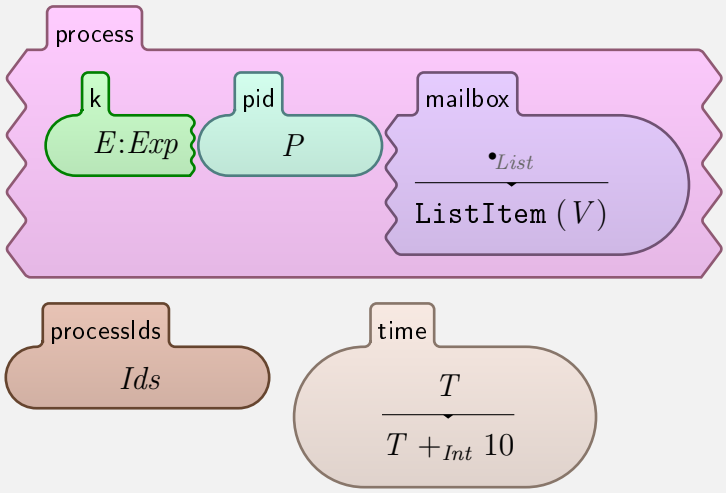
Itt is két szabályból áll a definíció. Ha már volt regisztrálva a *Name* változóban lévő atom, akkor a fent látható szabály illeszkedik. Az *undef* segítségével tudunk egy map-ben lévő kulcs értéket törölni a hozzárendelt értékeivel együtt. A második szabály átmenete természetesen a *badarg* futásidejű hibához vezet.

4.3.3. Üzenetküldés és -fogadás

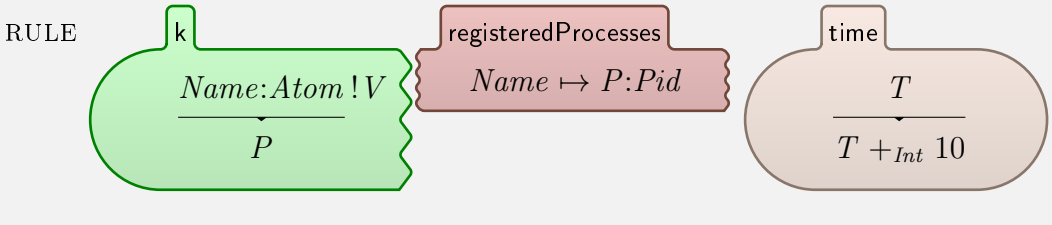
Ahogy már korábban említésre került a folyamatok üzenetekkel tudnak egymással kommunikálni. Egy üzenetet a ! operátorral küldhetünk, melynek mindkét operandusa kifejezés. Az első kiértékelése után folyamatazonosítónak vagy egy atomnak kell lennie, ám ebben az esetben előzőleg egy folyamatazonosítót kell regisztrálnunk a *register* függvénnyel különben *badarg* futásidejű hibát kapunk. Azonban akármilyen folyamatazonosítónak küldünk üzenetet, akár ténylegesen azonosít egy folyamatot akár nem, az üzenetküldés nem okoz futásidejű hibát. Az operátor visszatérési érték az üzenet.

Ehhez már bonyolultabb szemantikadefiníció tartozik 6 szabállyal. Ezek közül ötöt ismertetek, majd a *receive* bemutatása közben pedig az utolsót is.





Az üzenetküldéskor leggyakrabban használt eset szabálya látható felül. A P változóban lévő folyamatazonosító által meghatározott folyamatnak küldi el a V változóban lévő értéket. Az illeszkedés a P szerinti változó alapján történik, tehát illeszkedik a ! operátor kiértékelt első operandusára, és egy másik folyamat pid állapotára, emiatt látható a második process cella. Itt az átmenet a mailbox állapot listájának a végére fűzi a V üzenetet. A mailbox a folyamatokhoz tartozó postaláda állapota, a beérkező üzeneteket időrendben tároló lista. A feltétel második része érdekes csak most számunkra, ami azt mondja meg hogy az adott P folyamat tényleg létezik-e. Hasonló a szabály akkor is, ha egy folyamat saját magának akar üzenetet küldeni. Ekkor a szabály a saját pid állapotára illeszkedik, és az átmenet a saját üzeneteinek a végéhez fűzi az üzenetet. A harmadik szabály feltétele azt mondja ki, hogy az adott P folyamat nem létezik, így a k cella átmenete végbemegy, az üzenet lesz a cél, de semmi változás sem történik egyik folyamat postafiókjában sem.



A maradék két szabály a regisztrált névnek való üzenetküldésről szól. Fent látható, ha létezik regisztrált név a *registeredProcesses* állapotban, akkor a névből a hozzárendelt folyamatazonosítóba megy az átmenet. Itt feltétel sem szükséges, mivel az illeszkedés csak akkor történik meg, ha létezik az a bizonyos név-folyamatazonosító társítás. A *badarg* futásidejű hibát kiváltó szabály esetén ha a *registeredProcesssses* állapotban lévő mapben nincs hozzárendelve a névhez az adott kulcs, az azt jelenti, hogy nem regisztrált név, így az üzenetküldés *badarg* futásidejű hibát ad a k cella

átmenetében.

Az üzenet fogadása a *receive* kifejezéssel történik. A postaládában lévő üzenetekre sorban mintaillesztést végez. Először a legrégebbit veszi ki és felülről lefelé haladva elvégzi a *receive* ágaira. Ha az illesztés sikeres volt, és a hozzátartozó őrfeltétel is igaz, akkor kiértékeli a hozzátartozó ágat, és a kifejezés értéke az ág értéke lesz. Az üzenetet ekkor eldobja a postaládából. Ha nem volt sikeres mintaillesztés folytatja a másodikkal, és így tovább. A *receive* soha sem dob futásidejű hibát, vagyis ha egyik üzenetre sem volt sikeres a mintaillesztés, vagy sikeres volt, de a hozzátartozó őrfeltétel nem volt igaz, akkor a folyamat addig vár, amíg új üzenet nem érkezik, és kezdődik minden elölről. A hozzátartozó szemantikadefinícióban nem foglalkozunk az őrfeltételekkel, a szintaxis sem tartalmazza egyelőre.

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
end
```

A kifejezés tartalmazhat egy *after* klózt, ahol az *ExprT* kifejezés értéke egy nullánál nagyobb vagy egyenlő szám, vagy az *infinity* atom. Ellenkező esetben *badarg* futásidejű hibát kapunk. Ha van *after*, akkor a sikertelen mintaillesztések után ennyi milliszekundumig vár a folyamat új üzenetre. Ha az idő letelt, akkor az *after* utáni *BodyT* lesz kiértékelve, ami a *receive* visszatérési értéke lesz. Az *infinity* atom esete, ugyanolyan, mint ha nem lenne *after* klóz. Ez az eset az *ExprT* futásidőben való kiértékelésekor kap értelmet, hiszen így *after* klóz esetén is tudjuk szimulálni az *after* nélküli kifejezést.

Az *after* klóz szemantikadefiníciójához szükséges az eltelt idő ismerete. Sajnos a $\mathbb{K}$ keretrendszer jelenleg nem támogatja ezt semmilyen módon, így a szemantika részeként kellett megvalósítani. A már látott szemantikadefiníciókban feltűnhetett a *time* állapot, ami a program indítása óta eltelt időt jelenti milliszekundumban. Ám ez az idő relatív. Minden egyes utasítás növeli a time állapot értékét. Jelenleg ez egységesen 10 milliszekundum. Így egy fajta időmúlás érzetét kapjuk, ami természetesen, ahogy említettem relatív, vagyis nem azonos a való világban eltelt idővel.

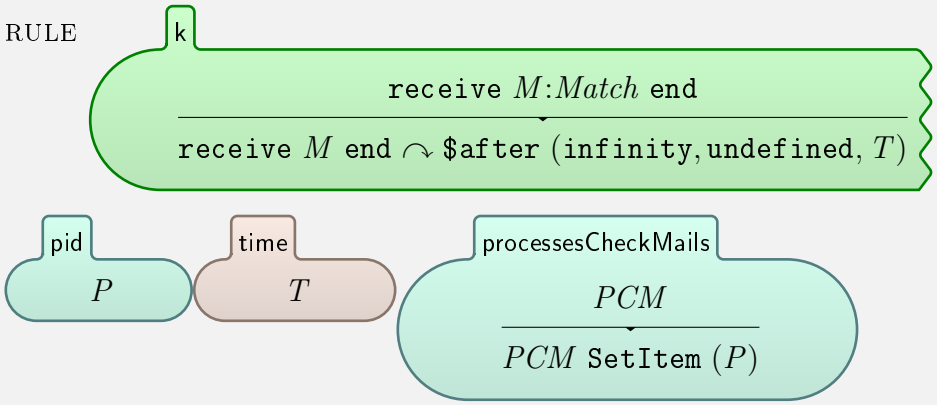
Főleg az *after* klóz miatt ez az egyik legbonyolultabb szemantikadefiníció, amit tartalmaz a diplomamunka. A különböző *receive* szintaxisok egységes kezeléséért lett

bevezetve az alábbi *After* fajta.

SYNTAX $\mathit{After} ::= \texttt{\$after}\ (\mathit{Exp}, \mathit{Exp}, \mathit{Int})$ [strict(1)]

Az első paraméter tartalmazza az *after* klózban meghatározott időt. Azért *Exp* fajta nem pedig *Int*, mert az *infinity* atomot is tartalmazhatja. A második paramétere az *after* klóz törzse, a harmadik pedig az az időpont, amikor a *receive* kiértékelése elkezdődött.

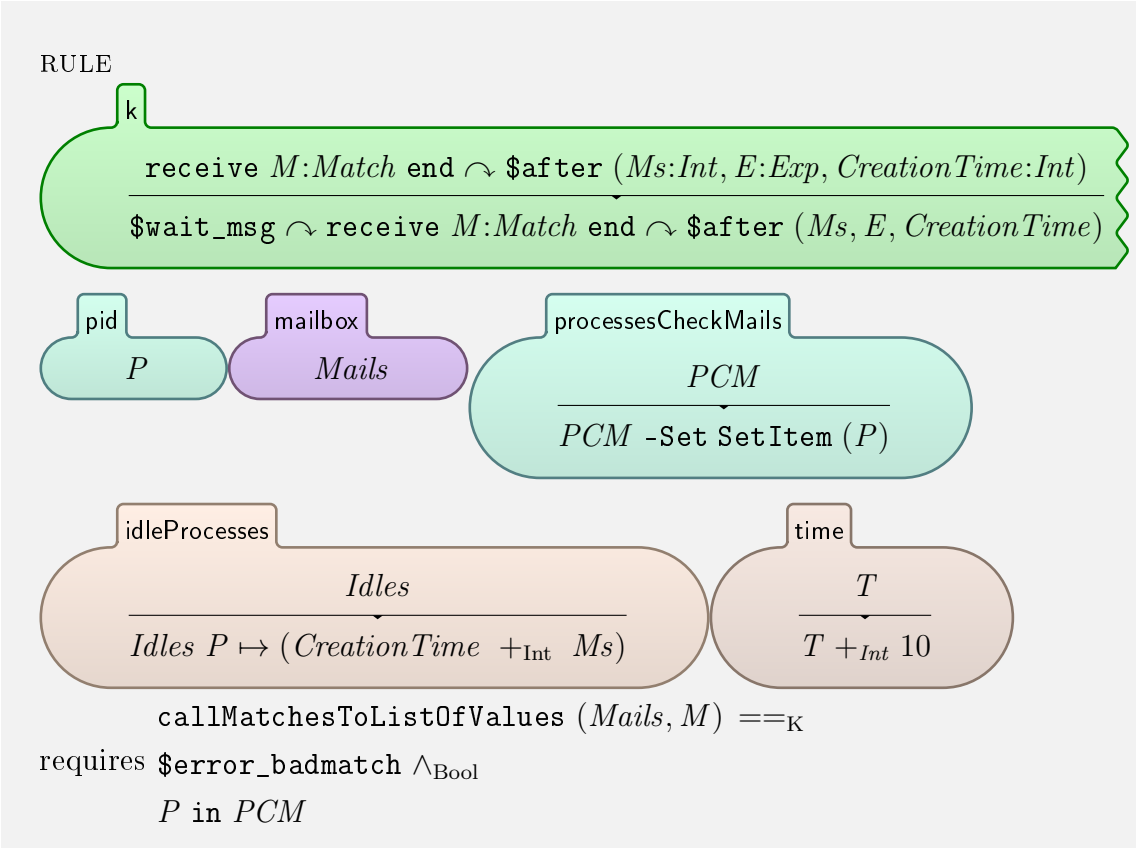
RULE



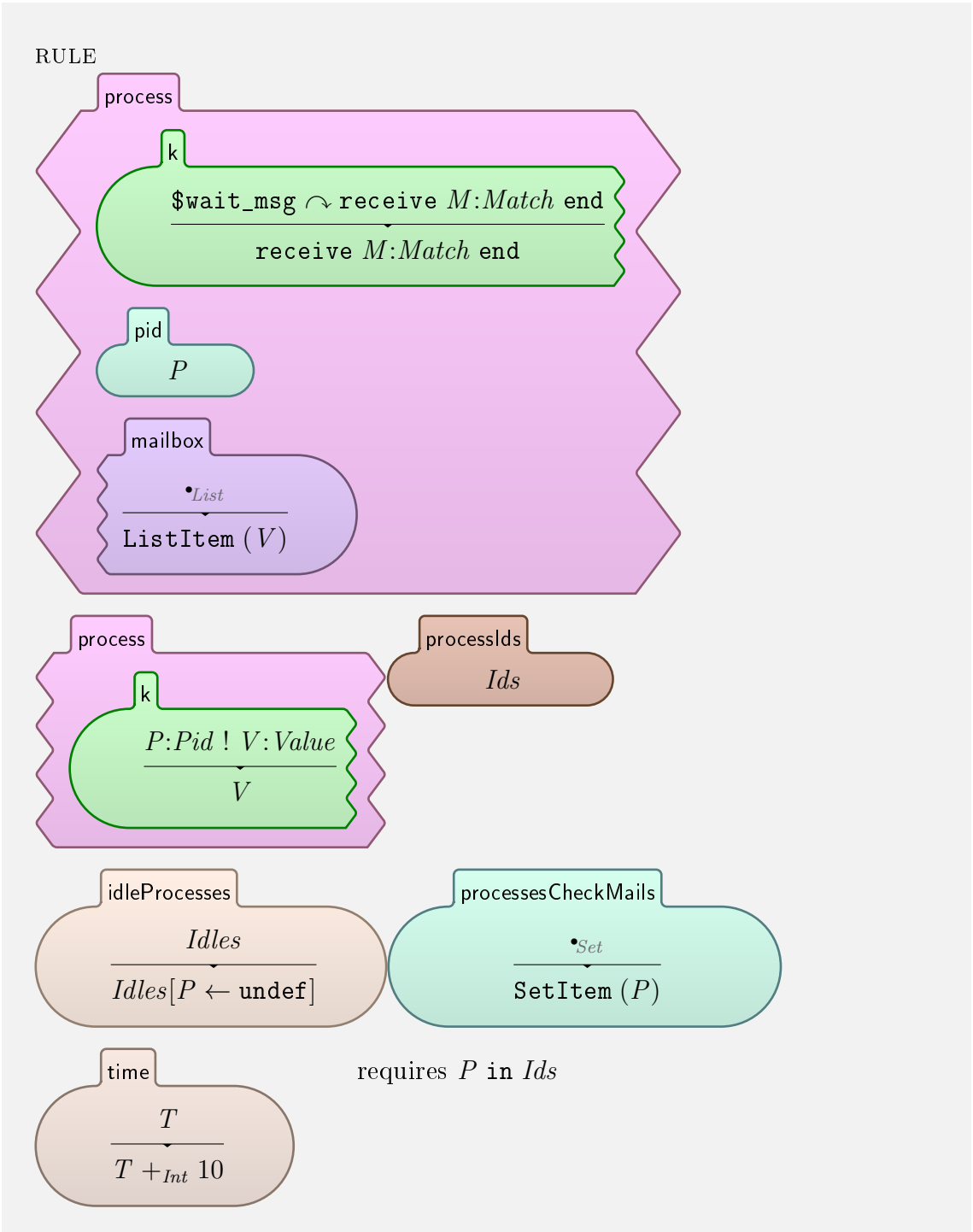
requires $\neg_{\mathrm{Bool}}(P\ \texttt{in}\ \mathit{PCM})$ [structural]

Minden *receive* híváskor egy strukturális átalakítást végző szabály illeszkedik először, ami a megfelelő átalakítást végzi. Fent látható a négy szabály közül az egyik. Ez az az eset, amikor nincsen *after* klóz. Látható a *k* cella átmenetében, hogy a számítási folyamba betesszük az előbb említett *$after*-t a megfelelő paraméterekkel. Itt az *infinity* atomot adjuk meg első paraméterben, második paraméterként pedig *undefined* atomot, mivel nincs *after* klóz, így nincs is kifejezés, amit ki kellene értékelni. Az idő állapotot lekérjük, és harmadik paraméterként adjuk át. Egy új állapotot fedezhetünk fel, ami a *processesCheckMails*. Ez azoknak a folyamatoknak a halmaza, amelyek éppen *receive* kifejezés esetén a mintaillesztést végzik, tehát nem várakoznak üzenetekre. Ez a szabály csak is akkor illeszkedik, ha még csak most ér a folyamat a *receive* kifejezéshez. Ennek oka az, hogy a strukturális átalakítások alatt lecsípjük az *after* részt, így a különböző szintaxisú *receive* kifejezések azonos formájúak lesznek a számítási folyamban, és egy már átalakított kifejezést nem akarunk még egyszer strukturálisan átalakítani, akár többször is. Mivel minden formájú *receive* kifejezés azonos lesz, a szemantikáját egységesen lehet definiálni. Természetesen már strukturális átalakítás során kiszűrjük a rossz argumentumú *after* klózt.

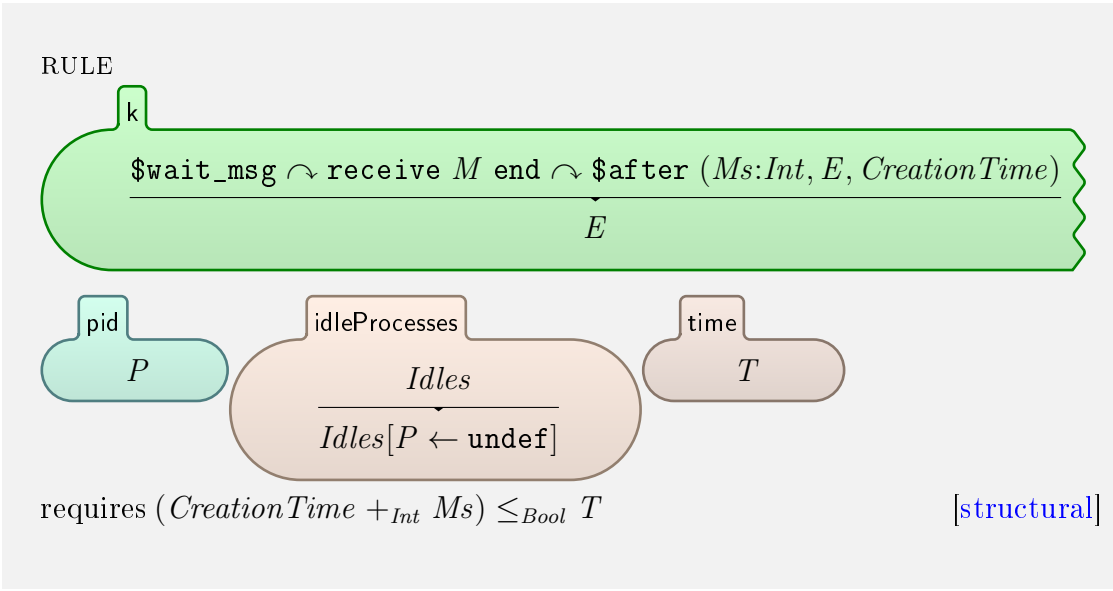
Üzenetfeldolgozásért három szabály felelős, egy amely sikeres mintaillesztés esetén illeszkedik, kettő pedig mikor sikertelen. Az egyik sikertelen esetet kezelő szabályról fogok részletesebben beszélni.



A feltételben lévő *callMatchesToListOfValues* végzi a mintaillesztést az üzenetekre. Ebben az esetben nem volt sikeres mintaillesztés, vagyis *$error_badmatch*-al tér vissza. A feltétel második része mondja ki, hogy csak azokra a folyamatokra illeszkedik ez a szabály, amelyeken a strukturális átalakítás lefutott, és készek rá, hogy megvizsgálják a postaládájukat. Hisz látható, hogy az előző szabályban és ebben a szabályban is a *k* cella eleje azonos, így ha ez a feltétel nincs, nem determinisztikusan illeszkedhet az egyik illetve a másik szabály. Mivel ez a szabály a postaládában lévő üzenetek mintaillesztését elvégzi, ezért az átmenet kiveszi a *processesCheckMails* állapotból a folyamat azonosítóját. Az átmenet a folyamatot várakoztató állapotba rakja. Ez két helyen is jelölve van. Az egyik a *k* cella átmenete, ahol a számítási folyamba berakja a *$wait_msg* konstanst, ami a későbbi szabályillesztésnél ad segítséget, illetve az *idleProcesses* állapotba berakja a folyamatazonosító-lejárati idő párt. Az *idleProcesses* állapot egy map, ahova a folyamatokhoz hozzárendel egy lejárati időt. Ezután a folyamat várakozik. Itt jön a képbe a hatodik szabálya a ! operátor szemantikadefiníciójának.

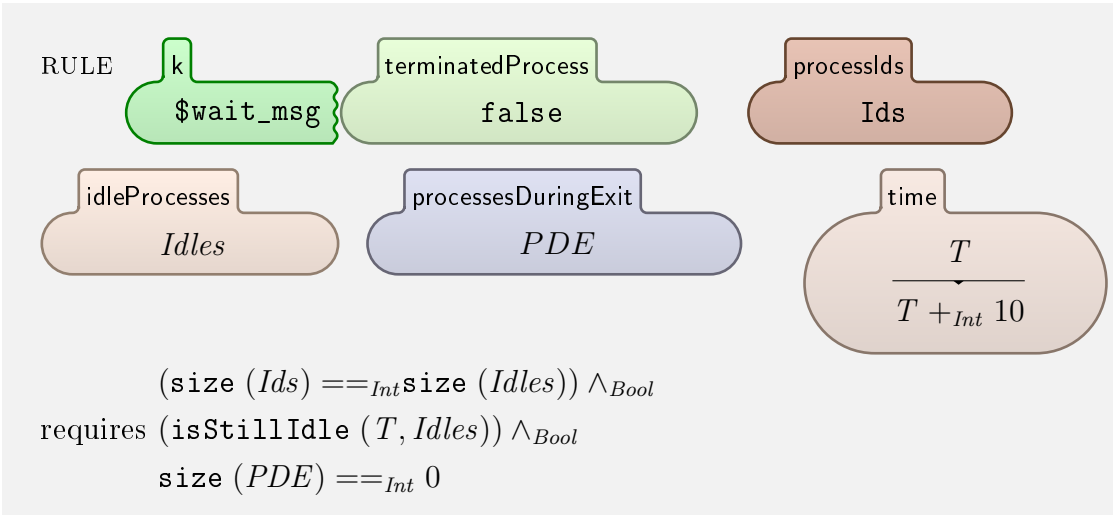


Mivel ez egy külön eset, külön szabály is tartozik hozzá. Ha egy folyamat várakozik, amit a *$wait_ msg* alapján is tudunk, akkor az üzenetküldés fogja beindítani, a *k* cella átmenetében, az újbóli feldolgozását a postaládában lévő üzeneteknek. Figyeljük meg az *idleProcesses* és a *processesCheckMails* állapotok változásait. A folyamat többé nem várakozó folyamat, hanem az üzeneteit vizsgáló folyamat. Ezen kívül még egy eset létezik (a kívülről jövő termináló üzeneten kívül) a várakozó folyamat állapotból való kiléptetésre, mégpedig ha az ideje lejárt.



A szabály csak várakozó folyamatokra illeszkedik. Ha a *$after* harmadik paramétere, vagyis a *receive* hívásának kezdetének ideje és az *after* klózban tartalmazott idő, ami az első paraméter, összege nagyobb, mint a *time* állapotban lévő idő, akkor többet nem vár a folyamat illeszkedő üzenetre, hanem az *after* klóz törzsét kiértékeli. Ekkor az *idleProcesses* állapotból kiveszi az adott folyamatazonosító-idő párost, ezzel jelezve, hogy már nem várakozó folyamat az alábbi.

Ezzel gyakorlatilag az üzenetküldés és -fogadás szemantikadefinícióját majdnem teljesen egészében lefedtük, viszont egy megválaszolatlan kérdés maradt. Mi történik akkor, hogy ha az összes folyamat üzenetre várakozik. Ha mindegyik *infinity* atommal lett meghívva, vagy *after* klóz nélkül, akkor a végtelenségig vár, a programunk megakadt, de ha legalább egy folyamat *after* klózában egy számot adtunk meg, akkor az idő letelte után az utána lévő kifejezést ki kell értékelnie. A probléma az, hogy mivel minden folyamat vár, tehát számítási lépés nem hajtódik végre, ezért nem fog telni az idő, vagyis nem növeli egy szabály sem az idő állapotban lévő értéket.



Itt fellelhető néhány állapot, melyek később lesznek ismertetve. A lényegi rész a feltételben és a k cellában található. Ha minden folyamat várakozik, akkor minden

folyamat k cellájában a *$wait_ msg* található. Ez a szabály bármelyik várakozó folyamatra tud illeszkedni. Ha az *idleProcesses* állapotban lévő elemek száma megegyezik a *processIds* állapot elemeinek számával és az éppen illeszkedett folyamatra hívott *isStillIdle* szemantikus függvény igazat ad, akkor az átmenet növeli a *time* állapot értékét, vagyis telik az idő. A feltétel első fele azt mondja ki, hogy ugyanannyi futó folyamatunk van, mint amennyi várakozik, tehát minden futó folyamatunk várakozik, a második pedig megvizsgálja, hogy a várakozó folyamatok közül van-e olyan, amelynek már letelt az ideje, vagyis az *after* rész kiértékelhető-e már.

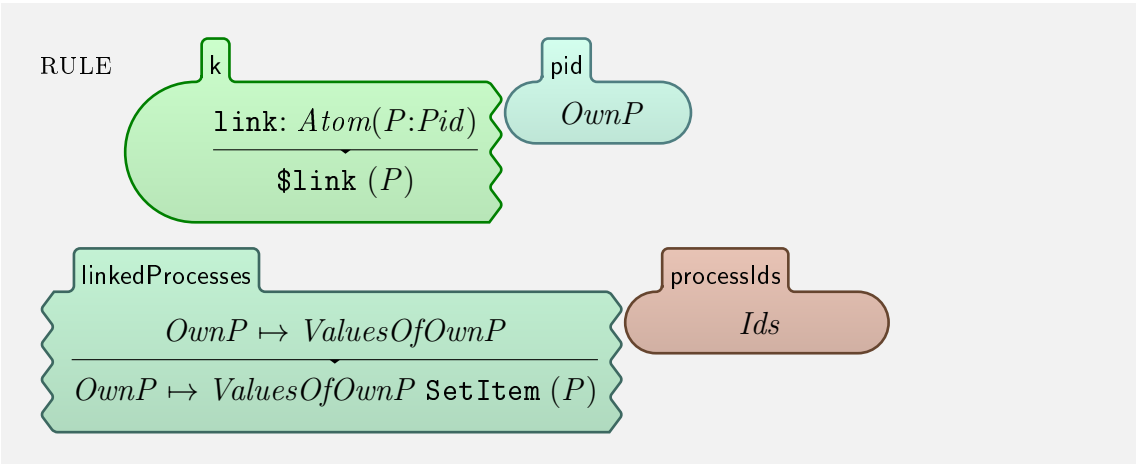
Ezzel körbe is jártuk az üzenetküldés és -fogadás szemantikadefinícióját. Ezt az összetett működést összesen 15 szabály segítségével adtuk meg.

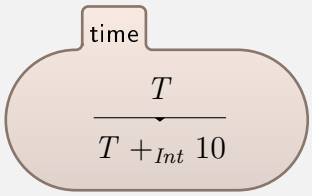
4.3.4. Folyamatok közötti kapcsolatok

A folyamatok között kialakítható egy kölcsönös kapcsolat a *link* függvénnyel, melyet párjával az *unlink*kel meg tudunk szüntetni. Ezek a kapcsolatok a folyamatok terminálásának kezelésére alkalmasak. Ha egy folyamat terminál, akár futásidejű hibával akár nem, akkor az összes hozzákapcsolt folyamatot értesíti erről. A *link* függvény mindig *true* atommal tér vissza. Egyetlen paramétere a folyamatazonosító, amely között a hívó folyamat kapcsolatot akar létrehozni. Ha már létezik, akkor nem fog létrehozni újat, mivel minden folyamat között csak egy kapcsolat lehet. Saját magával nem tud kapcsolatot létrehozni. Ha a paraméterben megadott folyamatazonosítóhoz nem tartozik futó folyamat, akkor két lehetőség létezik. Ha a *trap_ exit* flag értéke hamis, akkor a *noproc* futásidejű hibát adja, ha viszont igaz, a hívó folyamat egy üzenetet kap mégpedig az alábbi formában:

```
{'EXIT', Folyamatazonosito, noproc}
```

A *trap_ exit* flagről később lesz szó.





requires P `in` $Ids \wedge_{Bool} P = / =_{K} OwnP$

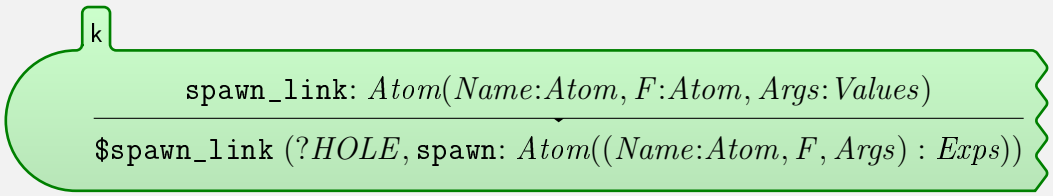
A folyamatok közötti kapcsolat kialakítása két lépésből áll. A *linkedProcesses* állapotban, ami egy map, tárolja a kapcsolatokat. Egy folyamatazonosítóhoz egy folyamatazonosítókat tartalmazó halmazt rendel. Először, ahogy fent látható, a *link* függvény a hívó folyamathoz rendeli a paraméterben átadott folyamatazonosítót. Az illeszkedés csak akkor sikeres, ha ehhez a folyamathoz létezik már ilyen kapcsolat. A hozzátartozó halmazt bővíti az első paraméterrel. Abban az esetben, ha már létezik kapcsolat a két folyamat között, ez a lépés gyakorlatilag nem fog semmi újat hozzáadni a *linkedProcesses* állapothoz, mivel a halmaz tulajdonság miatt nem lesz két ugyanolyan eleme, vagyis nem lesz két kapcsolat ugyanazon folyamatok között.

A második lépés is hasonló. Ahogy láthattuk a k cella átmenete a *$link*. Ez azért szükséges, mert jól el kell határolni az első folyamathoz való hozzárendelést a másodiktól. Az ok az, hogy előfordulhat az az eset, amikor az egyik folyamathoz már létezik kapcsolat, vagyis a *linkedProcesses* állapotban megtalálható, viszont a második folyamathoz még nem létezik, így új elemként kell beszúrni ebbe az állapotba. A *linkedProcesses* állapot redundánsan tartalmazza az információt a kapcsolatokról, viszont az ilyen fajta tárolás megkönnyíti az adatok kezelését.

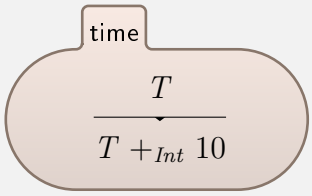
A *spawn_link* függvénnyel egy lépésben hozhatunk létre új folyamatot, és kapcsolhatjuk össze a hívó folyamattal.

SYNTAX $Exp ::= \texttt{\$spawn\_link}\ (Exp, Exp)$ [strict(2)]

RULE



[structural]



RULE $\dfrac{\texttt{\$spawn\_link}\ (?HOLE, P{:}Pid)}{\texttt{\$spawn\_link}\ (P, \texttt{link}{:}\ Atom(P))}$ requires $\neg_{Bool}$ `isPid` $(?HOLE)$

$$\text{RULE}\ \frac{\texttt{\$spawn\_link}\ (P{:}Pid, true)}{P}$$

$$\text{RULE}\ \frac{\texttt{\$spawn\_link}\ (P{:}Pid, \texttt{\$error\_noproc})}{P}$$

Fent látható a *spawn_ link* szemantikadefiníciója. Az ok, amiért ilyen bonyolult szabályhalmazzal lehetett megoldani ezt a definíciót az az, hogy a link függvény a *true* atommal tér vissza, míg a *spawn_ link* a létrehozott folyamat azonosítójával, emiatt nem lehet egyszerűen csak paraméterként átadni a *spawn* visszatérési értékét a *link* függvénynek. Egy speciális *Exp* fajta segítségével, ami a *$spawn_ link*, egy saját kontextust hoz létre a számítási folyamban, így csak az alsó három szabály tud illeszkedni rá. A második paramétere rendelkezik *strict* attribútummal, tehát a strukturális átalakítás után egyből elkezdi kiértékelni a *spawn* függvényt. A *?HOLE* egy ismeretlen változót jelöl, ami gyakorlatilag megfelel a joker karakternek, ám tudunk a szabályokban hivatkozni rájuk. Abban az esetben, ha kiértékelődött a függvény és a változó nem *Pid* fajtájú, akkor az első paraméter helyére berakja a második paraméterben értékként kapott folyamatazonosítót, tehát elmenti, hogy vissza tudjon térni vele, míg a második paraméterbe pedig berakja a *link* függvényt ugyanezzel a folyamatazonosítóval. Most már a *link* a második paraméter, ezért ki kell értékelnie a keretrendszernek, mielőtt foglalkozna a *$spawn_ link*-kel. Mivel az első paraméter folyamatazonosító, csak az utolsó kettő szabály tud illeszkedni a számítási folyamra. Előfordulhat az az eset, amikor a linkelés még be sem fejeződött, de a létrehozott folyamat már terminált is. Ebben az esetben a *link* függvény *noproc* futásidejű hibát dob, ez viszont nem releváns, mivel csak a hívó folyamat volt lassú, de a *spawn_ link* atominak tekinthető. Az utolsó szabály pont ezt küszöböli ki.

Az utolsó függvény az *unlink*, ami törli a kapcsolatot két folyamat között, ha létezik. Ha nem létező folyamat azonosítóját adjuk meg paraméterként, akkor sem kapunk futásidejű hibát, minden esetben a *true* atommal tér vissza. Ha a *trap_ exit* igazra van állítva, akkor még az alábbi üzenetet is be fogja rakni a hívó folyamat postaládájába.

```
{'EXIT', Id, _}
```

A következő alfejezet a folyamatok terminálásának viselkedését és szemantikadefinícióját mutatja be, mellyel még világosabbá válik a folyamatok közötti kapcsolatok jelentősége.

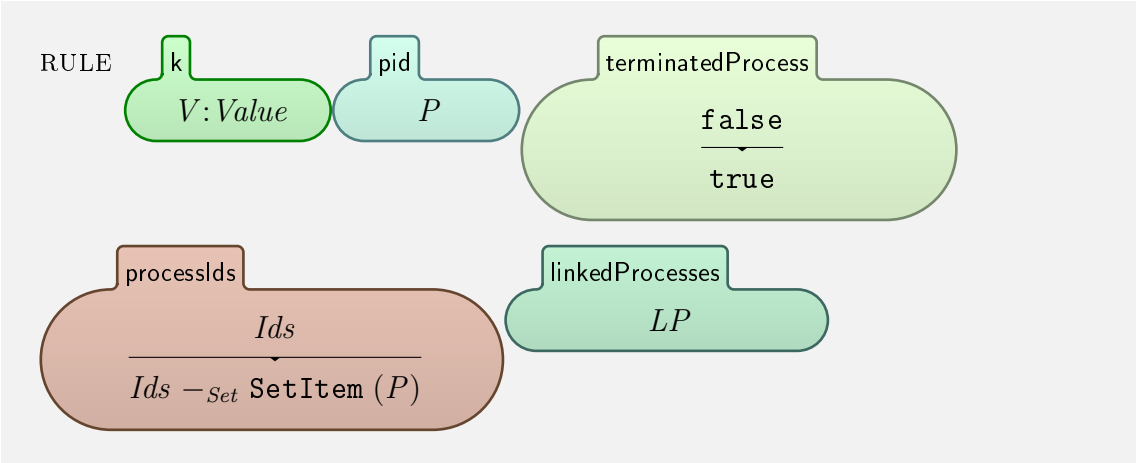
4.3.5. Folyamatok terminálása

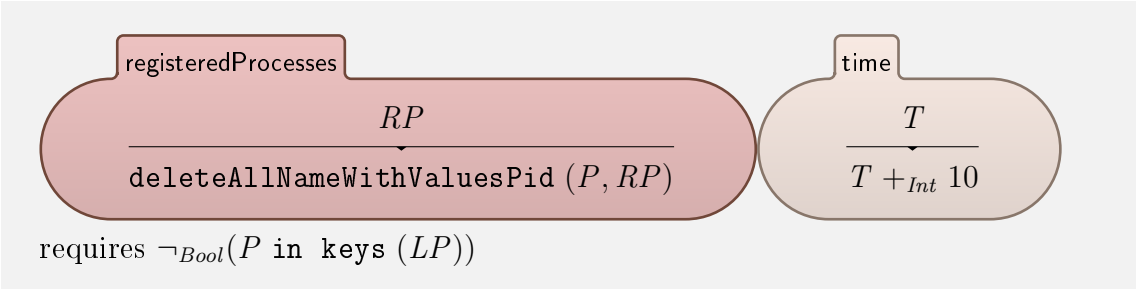
A folyamatok változatos formában képesek terminálni. Ennek egyik fajtája az egy és kettő aritású *exit* függvény. Az egy paraméteres változat a hívó folyamatot terminálja, mégpedig a paraméterben megadott indokkal. A két paraméteres változattal a második paraméterben megadott folyamatazonosító által identifikált folyamatot terminálhatjuk. Véget érhet még futásidejű hibával is, illetve gond nélkül, mikor már nincs mit kiértékelni. Itt jönnek képbe a folyamatok közötti kapcsolatok.

Amikor egy folyamat terminál, akkor a vele összekötött folyamatoknak küld egy szignált a terminálás okával. Ezek a folyamatok többféleképpen reagálhatnak erre a szignálra. Függ a terminálás okától és a *trap_exit* flag értékétől is. Erről a flagről már olvashattunk korábban. A két paraméteres *process_flag* függvénnyel állítható az értéke. Első paraméterként azt a flaget jelölő atomot adjuk meg, aminek értékét szeretnénk megváltoztatni, másodikként pedig az értéket. Visszatérési értéke a flag előző értéke lesz. Ezzel a függvénnyel beállított flageket a *processFlag* állapot tárolja, ami egy map, mégpedig a flagekhez rendelt értékek. Ha nem létező flagnek akarunk értéket adni, a függvény *badarg* futásidejű hibát dob. Ha a *trap_exit* értéke *false*, akkor ha exit szignált kap, melynek oka nem *normal* atom, akkor a folyamat is terminál és értesíti erről egy ugyan ilyen szignállal a hozzá kapcsolt folyamatokat. A *normal* atom esetén nem csinál semmit. Ha az érték *true*, akkor ha a kilépésnek oka nem *kill*, a szignált üzenetként fogja megkapni, az alábbi formában:

```
{'EXIT', Folyamatazonosito, TerminalasOka}
```

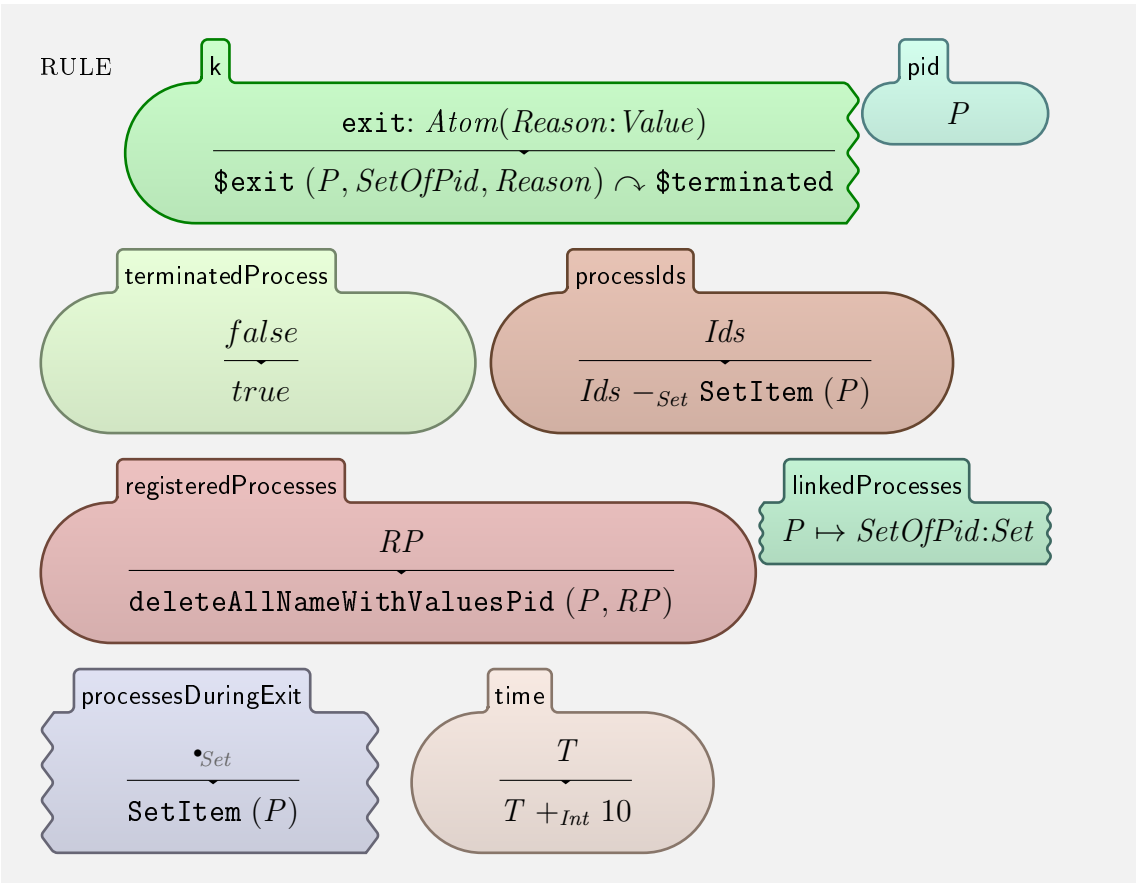
A *Folyamatazonosito* annak a folyamatnak az azonosítója, amely küldte a szignált, a *TerminalasOka* pedig az ok, amiért a folyamat terminált. Ha ez az ok *kill*, akkor a folyamat nem tudja üzenetként fogadni a szignált, úgy viselkedik, mint ha ez a flag *false* értékre lenne állítva.





Fent látható a terminálásnak a legegyszerűbb esete. A feltételből látszik, hogy ez a folyamat nincs összekapcsolva egy folyamattal sem, emiatt nem küld exit szignált. Az átmenet gyakorlatilag eltünteti a folyamat azonosítóját az összes állapotból, ahol megjelenik. Így ha volt hozzá regisztrált név, azt törli, a futó folyamatok halmazából is kiveszi a folyamatazonosítóját, és a *terminatedProcess* állapotát hamisra állítja. Ez az állapot jelzi az összes folyamatnál, hogy terminált-e. Ez egy redundáns információ, mivel ha a *processIds* nem tartalmazza a folyamatazonosítót, ugyanazt jelenti.

Ha a folyamat természetes módon terminál, és tartozik hozzákapcsolt folyamat, akkor annak a viselkedése teljes mértékben megegyezik az *exit(normal)* hívással. Ha a folyamat futásidejű hibát okoz, akkor a terminálása megegyezik a *exit(Reason)* hívással, ahol a *Reason* a futásidejű hiba. Ezekben az esetekben a k cellához tartozó átmenetek ezeket a függvényeket teszik bele a számítási folyamba.

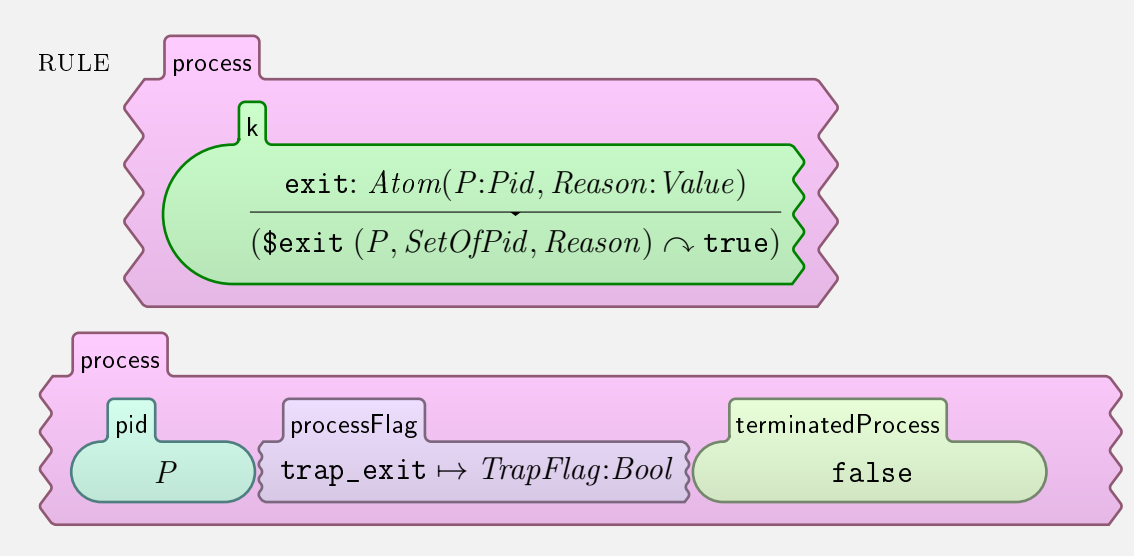


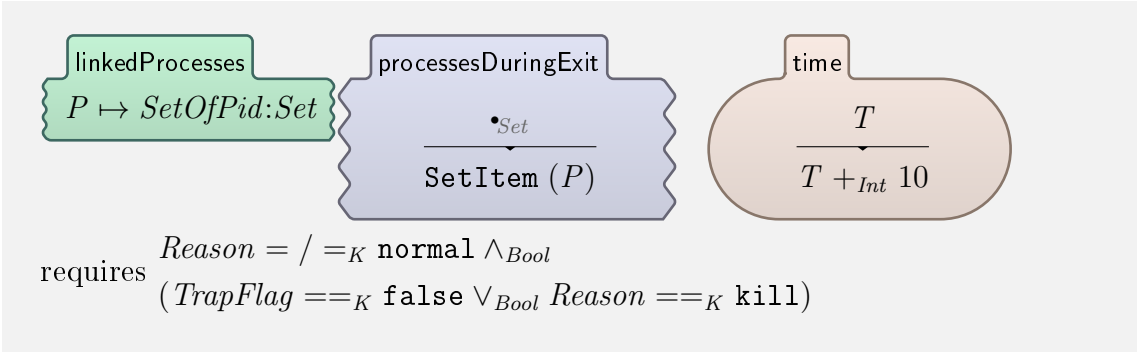
A fent látható szabály az egy paraméteres *exit* függvény szemantikadefiníciójához tartozó szabályok egyike. Abban az esetben, ha össze van kapcsolva folyamatokkal,

akkor kiolvassa ezek azonosítóját a *linkedProcesses* állapotból. A *k* cella átmenetében látható egy új *Exp* fajta, a három paraméteres *$exit*, ami egy környezetet ad a folyamat szignál küldéseihez. Első paramétere a folyamat, ami a szignált küldi, második paramétere azon folyamatazonosítók halmaza, mellyel az első paraméterben lévő folyamat össze van kapcsolva, utolsó paramétere pedig a terminálást kiváltó ok. A *$exit* környezet azért lett létrehozva, hogy egységesen lehessen kezelni az egy és a két paraméteres *exit* függvényeket. Láthatjuk hogy az átmenet itt is eliminálja a folyamatazonosítót a lehetséges állapotokból, a *processesDuringExit* állapotot kivéve. Ez az állapot azért jött létre, mivel egy lépésben nem kivitelezhető az összes szignál küldés és a folyamat terminálása. Ezzel el tudjuk kerülni, hogy egy éppen terminálás alatt lévő folyamat szignál küldéseit összezavarja.

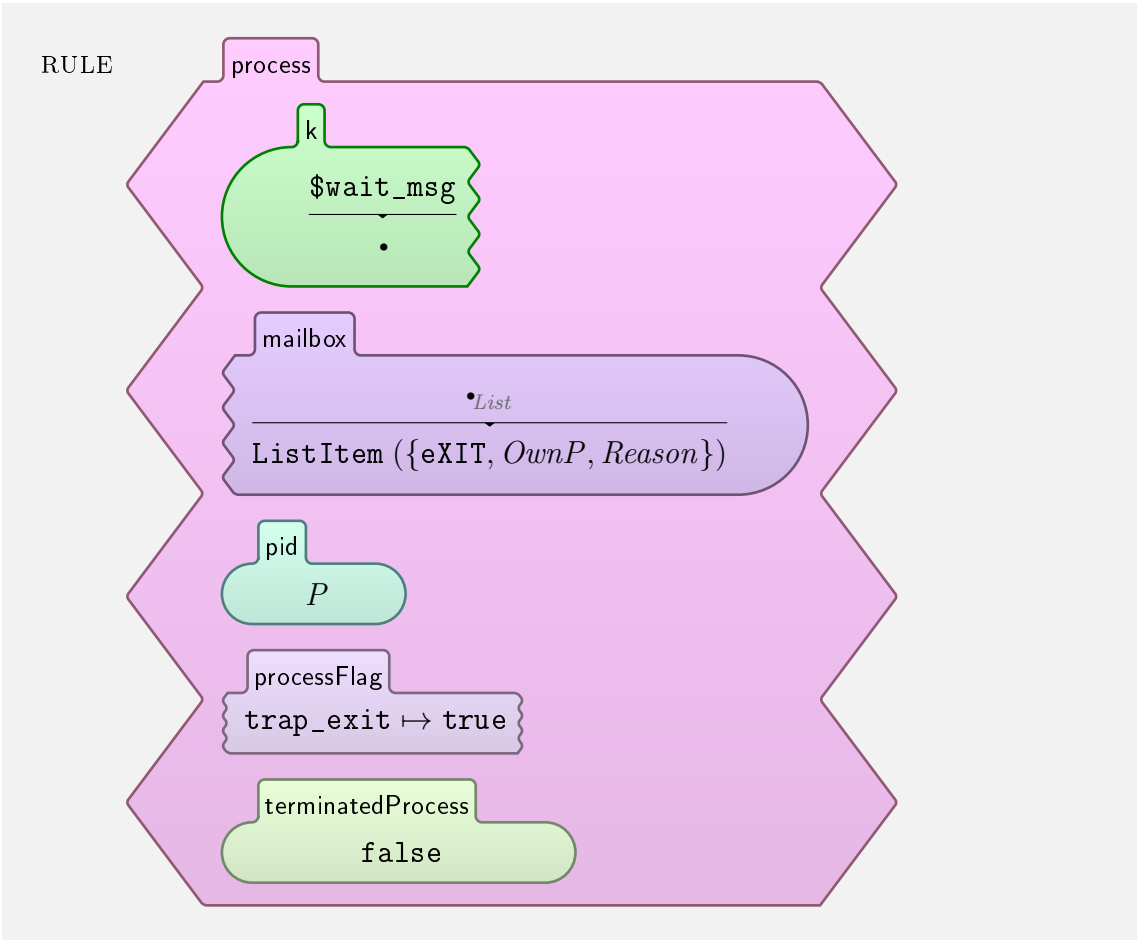
Érdemes megfigyelni a *k* cella átmenetét. A *$exit* után egy *$terminated* kerül a számítási folyamba. Ennek oka, hogy mindenféle feltétel nélkül akarjuk megállítani a további kiértékelést. Miután a *$exit* környezet eliminálása megtörténik, a maradék kód még ottmarad a *k* cellában. Ahhoz hogy ne értékelje tovább, minden egyes szabálynak tartalmaznia kellene egy olyan feltételt, ami azt állítja, hogy a *processIds* állapotban benne van a folyamat azonosítója. Ezzel a trükkel viszont nem kell a többi szabályt bővíteni, hiszen a *$terminated* miatt nem fognak illeszkedni a konfigurációra.

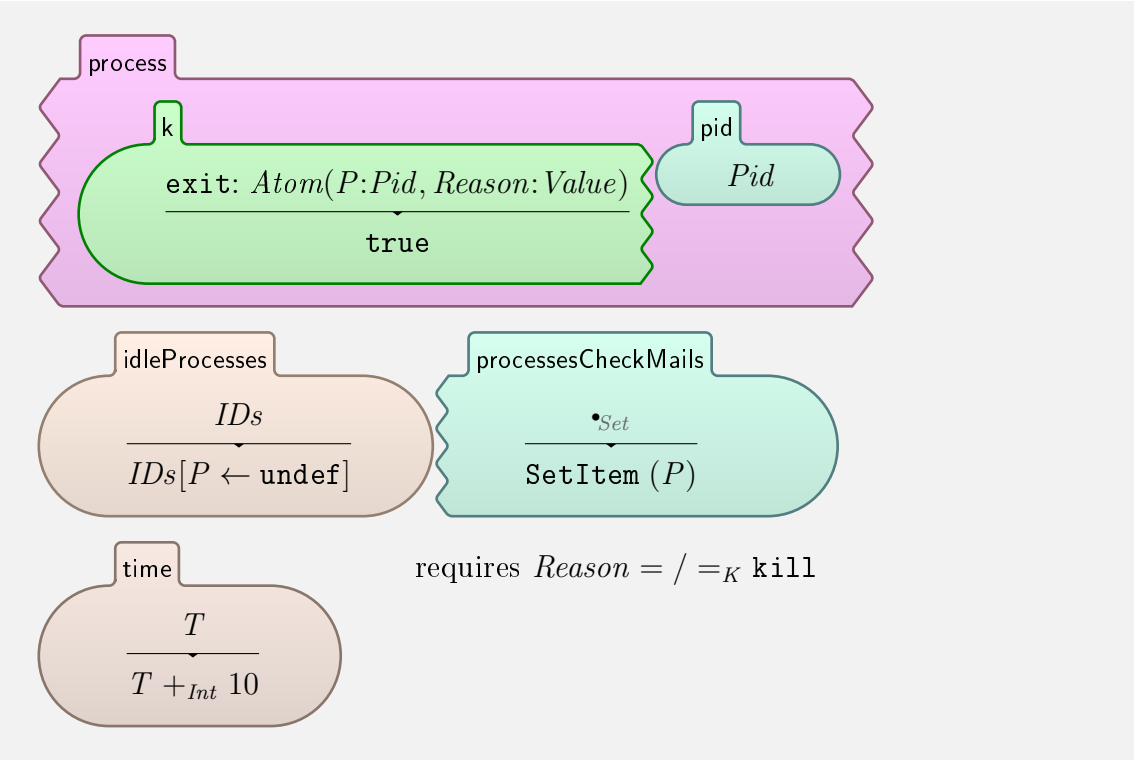
A két paraméterű *exit* függvénynél már bonyolultabb szabályok jelennek meg, mivel ott még azt is kell figyelni, hogy a *trap_exit* flag értéke mire van beállítva. Ezenkívül küldheti saját magának is az üzenetet. Itt a szabályok halmaza két nagy részre osztható. Az egyik mikor kell exit szignált küldenie, a másik pedig mikor üzenetet kell küldeni a folyamatnak.





Ebben a szabályban az egyik folyamat terminál egy másikat. Ha nem a *normal* okkal, és a folyamat *trap_ exit* flagje hamis vagy az ok a *kill* atom, akkor a folyamatot terminálja, és mivel van hozzákapcsolva folyamat, a k cellában az átmenet szintén a *$exit* környezetet hozza be, és elkezdődik az exit szignál küldése. A szemantikadefiníció többi szabálya is hasonló. Mind a feltételben különbözőek, és attól függően kezelik a folyamat terminálását. Érdekes szabály még az alábbi is.

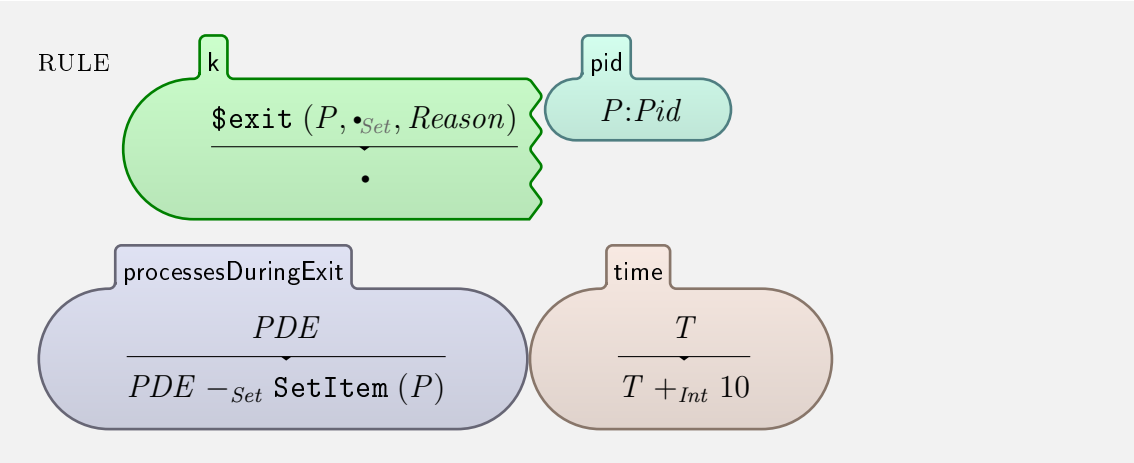




Ez azt az esetet írja le, mikor egy folyamat éppen *receive* kiértékelése közben van, üzenetre vár, és a *trap_ exit* flaghez pedig a *true* atom van hozzárendelve. Ekkor nem csak az üzenetet kell elküldeni, hanem azért is felelős, hogy a folyamat elkezdje újra elölről átnézni a postafiókját. Emiatt a folyamat k cellájában a *$wait_ msg*-t eliminálja. Ha újra a folyamatra esik a kiértékelés, akkor elkezdi ellenőrizni a postafiókját. Természetesen ha a terminálló ok a *kill* atom, akkor ez a szabály nem alkalmazható.

Még sok szabály van, amit itt külön nem részletezek. Ilyenek például a már említett esetek, mint mikor a saját folyamatazonosítóját adja meg paraméterként, vagy a terminálás oka *normal* atom, vagy az adott folyamatazonosító nem identifikál semmilyen folyamat. Összesen tíz darab szabály tartozik a két paraméterű *exit* függvény szemantikadefiníciójához. Már csak egy kis részlet maradt hátra a *$exit*-hez tartozó szabályok, melyek a szignál küldését bonyolítják le.

Ezek a szabályok gyakorlatilag egy az egyben megegyeznek a két paraméteres *exit* függvénnyel. A második paraméterben lévő halmazt járják végig, és minden egyes folyamatot módosítják feltételektől függően, mint például a terminálló ok, és a *trap_ exit* flag. Mondhatni egy rekurzív függvényt szimulál, melynek a kilépési pontja az alábbi szabály.



Ha az összes hozzákapcsolt folyamatnak elküldte az exit szignált, vagyis a második aktuális paramétere üres halmaz, akkor a folyamat végleg terminálhat. Előzőleg már a *processIds* állapotból ki lett véve a folyamatazonosítója, illetve minden állapotból kikerült még a terminálás bekövetkeztekor, vagy az *exit* hívásakor. Tehát már csak a *processesDuringExit* állapotból kell kivenni a folyamatazonosítóját, ezzel jelezve hogy befejezte az exit szignálok küldését.

Összesen 21 szabály kapcsolódik a terminálással kapcsolatos szemantikához. A konkurens résznyelvet leíró szemantikadefiníció végül 61 szabályt és 9 szemantikus függvényt tartalmaz.

4.4. Tesztelés

Egy bonyolult és összetett szemantikadefiníciót ismerhettünk meg az előző fejezetben. A függvények sokszor egymástól függnek, ilyen például a folyamatok közötti kapcsolatok és a terminálás. A definíció írása alatt könnyedén lehetett úgy bővíteni a szabályhalmazt és az állapotteret, hogy a már megírt szabályok nem adták vissza megfelelően a manuálban leírtakat. Emiatt a definíció írásának jól meghatározott ciklusai voltak.

A ciklus első lépése a manuál értelmezése volt, a nem egyértelmű részek megvitatása, ezekre válasz keresése. Erlanghoz egy széles körben elterjedt implementáció létezik [Erl], amely jó referenciát adott kérdéses esetekben. Második lépésként az összegyűjtött adatok alapján apró példakódokat írni, mellyel a függvény egy-egy specifikus működését lehetett ellenőrizni. Harmadik lépés pedig maga a szabályhalmaz megírása, amit az előző fejezetben láthattunk is. Abban az esetben, ha a megírt részekhez tartozó tesztek megbuktak, negyedik lépésként össze kellett hangolni a szabályokat, esetleg a teszteket módosítani, hogy újra zöld legyen az út.

A második lépésben segítséget nyújtott a $\mathbb{K}$ keretrendszer tesztelő egysége. Ha futtatunk egy programot a *krun* paranccsal, akkor a végkonfiguráció lesz az eredmény, ha terminált a program. Ebből lehet mintaillesztéssel információt nyerni.

```
count_to_ten() -> do_count(0).
do_count(10) -> 10;
do_count(Value) -> do_count(Value + 1).
---
count_to_ten().
```

Legegyszerűbben egy példán keresztül lehet bemutatni a működését. Vegyük a fenti programot. A *count_to_ten* függvénnyel a rekurzív hívás szemantikáját lehet tesztelni. A visszatérési értéke mindig a tízes szám lesz, abban az esetben ha a szemantikadefiníciók helyes. Így a végkonfiguráció k cellájában a tízes értéknek kell szerepelnie. Helyes működés esetén az alábbi végkonfigurációt kapjuk:

```
<T>
  <processes>
    <process>
      <k> 10 </k>
      <env> .Map </env>
      <pid> < 0 . 0 . 0 > </pid>
      <mailbox> .List </mailbox>
      <processFlag> trap_exit |-> false </processFlag>
      <terminatedProcess> true </terminatedProcess>
    </process>
  </processes>
  <processIds> .Set </processIds>
  <processesCheckMails> .Set </processesCheckMails>
  <idleProcesses> .Map </idleProcesses>
  <processesDuringExit> .Set </processesDuringExit>
  <registeredProcesses> .Map </registeredProcesses>
  <linkedProcesses> .Map </linkedProcesses>
  <time> 330 </time>
  <defs> do_count |-> ( ListItem ( { 10 , .Exps } -> 10 ) ↩
     ListItem ( { Value , .Exps } -> do_count ( ( Value + ↩
     1 ) , .Exps ) ) ) count_to_ten |-> ListItem ( { ↩
     .Exps } -> do_count ( 0 , .Exps ) ) </defs>
</T>
```

Ez egy nagyon jó példa arra, hogy egy ilyen kis program esetén is ugyanazzal a hatalmas állapottérrel dolgozunk, mint egy összetettebb, több folyamatot tartalmazó programnál. Leegyszerűsíthetjük, kiemelhetjük mintaillesztéssel a számunkra hasznos részt a *–pattern* kapcsolóval és egy utánalévő mintával. Ebben az esetben a k cella végállapotára vagyunk kíváncsiak, és csakis abban az esetben, ha az *Value* fajtát tartalmaz. Ekkor az alábbi mintát kell megadni:

```
<k> V:Value </k>
```

A minta a *k* cellára illeszkedik. Ha több folyamat lenne a végkonfigurációban, akkor az összes *k* cellára illeszkedne, ami csak egy *Value* fajtát tartalmaz, és az összes eredményt visszaadná kimenetként. Lefuttatva a programot ezzel a mintával eredményként az alábbit kapjuk:

```
Search results:

Solution 1:
V:Value -->
10
```

Ez már könnyebben olvasható, mint a fenti teljes végkonfiguráció. Ha ezt a kimenetet elmentjük a programot tartalmazó fájl nevével a végére illesztve a *.out* kiterjesztést, akkor egyszerűen automatizálhatjuk a tesztjeinket. Ezt a *ktest* paranccsal és egy hozzátartozó *config.xml* fájllal tehetjük meg, ahol megadhatjuk, hogy az egyes programokra milyen mintát illesszen, és az eredményt összehasonlítja a *.out* fájlban lévővel. Ennél természetesen sokkal bonyolultabb mintákat is lehet konstruálni, és tesztelés során kellett is.

Azonban a rendszernek vannak hiányosságai. Például, ha egy map tartalmára vagyok kíváncsi, akkor ott számít a sorrend. Sokszor előfordult, hogy *kompile* parancsot használva ugyanazt az eredményt kaptam meg, de a map sorrendje megváltozott. Ezen felül az idő ellenőrzése is sokszor bajos volt, mivel volt hogy néha kevesebb, néha kicsit több volt, főleg ha a szabályhalmaz bővítve lett. Mindezek ellenére nagyon hasznos eszköznek bizonyult munkám során.

A konkurens rész összesen 38 tesztet tartalmaz. Minden teszt esetén a mintával az éppen számunkra érdekes információt nyerjük ki a végkonfigurációból, amiből kiderülhet, hogy az adott kódrészlet helyesen futott-e le. Felmerülhet a kérdés, hogy ezek a tesztek ténylegesen a helyes működést ellenőrzik? Minden egyes teszt írásakor a manuálra és az implementációra építettem, ahogy a szemantikadefiniálásakor is.

5. Összefoglalás

A diplomamunkában olvashattunk a keretrendszerről, mely az operációs szemantikában megfogalmazott formális definíciókra támaszkodva képes interpretert készíteni, áttekintettük a meglévő szemantikadefiníciót, bevezetést kaphattunk az Erlang folyamatok világába, ezután a konkurens résznyelv szemantikájába ástuk be magunkat megvalósítva a $\mathbb{K}$ keretrendszer segítségével, és végül az ehhez tartozó tesztekről hallhattunk.

A témabejelentőben kitűzött célok nagy részét sikerült teljesítenem. Hiányzik az *erlang:error* egy és két paraméterű változatának definíciója. Azonban a működésük hasonló a már definiált egy paraméterű *exit* függvényhez. Az *erlang:error* csupán kiegészíti a működését azzal, hogy nem csak a terminálás okát, hanem az aktuális folyamat vermét is elküldi a szignálban.

A monitorozásnak maradt még ki a szemantikadefiníciója. Különbség a folyamatok közötti kapcsolatokkal, hogy ez csak egyirányú, tehát tényleg egy folyamat monitoroz egy másikat és fordítva nem. Ennek megvalósítása könnyebb, mint a folyamatok közötti kapcsolatoké, az egyirányúság miatt. Mindig üzenetet küld a monitorozó folyamatnak ha terminál, nem foglalkozik a *trap_exit* flaggel és nem terminál a megfigyelő folyamat, ha a monitorozott leáll.

A cél a teljes formális nyelvdefiníció megírása, ami hosszadalmas folyamat. A jelen diplomamunka az elosztott esetekkel nem foglalkozott, amivel a *CONCURRENT* modul fog bővülni. Felmerül a kérdés, hogy a hardveres okokból eredő üzenetek késésének kezelése hogyan jelenik meg a szemantikadefinícióban. Ezenkívül külön függvények is tartoznak az elosztott programozás témájához.

Időközben egy másik ágon elkészült az Erlang modulra bontás definíciója is, amit majd össze kell fésülni az én munkámmal, figyelve milyen elemek kerültek be a konfigurációba, hogy a hasonló vagy ugyanazt kifejező állapotok eliminálva legyenek. Azonban a meglévő szemantikadefinícióval is elkezdődhet kisebb programok verifikálása. Igyekeztem a legnagyobb precizitással és körüljárással értelmezni a függvények pontos működését, majd formális szabályok formájába önteni, melynek eredménye ez a diplomamunka lett.

Irodalomjegyzék

[Erl] Erlang Programming Language. Elérve: 2018-05-09. URL: https://www.erlang.org/downloads.

[Erl16] Erlang/OTP 20.2 - Reference Manual, 2016. Elérve: 2018-05-09. URL: http://www.erlang.org/doc/.

[Fra] K Framework. Hivatalos weboldal. Elérve: 2018-05-09. URL: http://www.kframework.org/.

[Fre16] Fred Hébert. Learn You Some Erlang for Great Good, 2016. Elérve: 2018-05-09. URL: http://learnyousomeerlang.com/.

[Gri12a] Grigore Rosu. K and Matching Logic, 2012. Elérve: 2018-05-09. URL: http://www.kframework.org/images/archive/9/9b/20111119040515!K-and-Matching-Logic-Latest.pdf.

[Gri12b] Grigore Rosu, Andrei Stefanescu, Stefan Ciobaca, Brandon M. Moore. Reachability Logic, 2012. Elérve: 2018-05-09. URL: http://fsl.cs.illinois.edu/pubs/rosu-stefanescu-ciobaca-moore-2012-tr.pdf.

[Gri15] Grigore Rosu. Matching Logic - Extended Abstract, 2015. Elérve: 2018-05-09. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5185/pdf/5.pdf.

[Gri16] Grigore Rosu. ETAPS 2016 - K: a semantic framework for programming languages and formal analysis tools. https://www.youtube.com/watch?v=3ovulLNCEQc. Elérve: 2018-05-09. Április 2016.

[Hor15] Horpácsi Dániel. Formális Szemantika előadás, 2015.

[Pet06] Peter D. Mosses. Formal Semantics of Programming Languages, 2006.