Erlang folyamatok és a köztük lévő kapcsolatok felderítése

Bozó István, Tóth Melinda

Eötvös Loránd Tudományegyetem, Informatikai Kar

{bozo_i, toth_m}@inf.elte.hu

1. Bevezetés

Az Erlang [3, 4] egy dinamikusan típusos funkcionális programozási nyelv, amely masszívan párhuzamos, illetve amelyet elosztott alkalmazások fejlesztésére terveztek.

A RefactorErl [1] nevű eszköz egy statikus elemző és transzformáló keretrendszer, melynek egyik célja Erlang programok megértésének támogatása. Ehhez hozzátartozik az elosztott és párhuzamos programok elemzése is. Azonban ezek elemzése jóval nehezebb, mint a szekvenciális programok elemzése, főként egy olyan dinamikus nyelv esetén, mint az Erlang. Sok esetben statikus módszertannal a dinamikus információ töredéke számítható ki (becsülhető meg) fordítási időben; ám az is igaz, hogy ez még mindig több lehet és gyorsabban számítható, mint egy programozó által kézzel összeszedhető információ.

A célunk az volt, hogy a statikus elemzés lehetőségeinek megfelelően adjunk egy olyan folyamat elemzést, amely segítségével az Erlang programozási nyelven írt párhuzamos és elosztott programokból kinyerhető azok kommunikációs modellje. Ez a modell nagyban elősegíti a programozók számára a kódmegértést, hiszen ennek segítségével képet alkothatnak arról, milyen folyamatok vannak egy rendszerben, és ezek hogyan kommunikálnak.

A RefactorErl keretrendszert bővítettük egy újabb elemzéssel, amelylyel előállítható egy könnyen átlátható statikus kommunikációs gráf.

2. RefactorErl

A RefactorErl [2] egy statikus elemző és transzformáló keretrendszer. Ezen keretrendszer Erlang programok elemzését és transzformálását (refaktorálását) teszi lehetővé.

A keretrendszer egy szemantikus programgráfban (SPG [5]) reprezentálja a programokat, amely három rétegből épül fel:

- `Lexikális`
- `Szintaktikus`
- `Szemantikus`

A gráf egy $SPG = (N, A_N, A_V, A, T, E)$ hatossal adható meg, ahol

- N – a gráf csúcsainak a halmaza
- A_N – attribútum nevek halmaza
- A_V – lehetséges attribútum értékek halmaza
- $A : N \times A_N \to A_V$ – csúcs címkéző parciális függvény
- T – él címkék halmaza
- $E : N \times T \times N_0 \to N$ – élcímkéző parciális függvény rendezéssel

Az elemzések során az ebben a gráfban tárolt információt használjuk majd fel, illetve a gráfhoz új éleket és csúcsokat is adunk meg majd.

A lexikális elemző a forráskódból előállítja a tokeneket, amelyekből a szintaktikai elemző segítségével egy szintaxisfa épül.

A szemantikus elemzők különféle információkkal bővítik ezt a szintaxisfát:

- változók kötési és hivatkozási helye
- függvények definíciója és ezek alkalmazásai
- függvényhívási információ
- rekord és rekordmező információk
- stb.

A szemantikus elemzők nagy része inkrementális, azaz ha változás történik a gráfban, akkor ezeket lokálisan kezeli és csak a szükséges gráfrészletekben állítja helyre az információkat. A RefactorErl további, nem inkrementális elemzéseket is tartalmaz, mint például a dinamikus függvényhívás elemző. Ezek olyan információkat használnak és állítanak elő, amelyeket minden módosításnál újra elő kell állítani a teljes gráfra.

Az általunk megvalósított elemzés segítségével a keretrendszerbe betöltött programok statikus kommunikációs modelljét lehet előállítani, melyhez a szintaktikus és szemantikus elemzők által előállított információkat használjuk fel [6, 7]. Az elemzés nem inkrementális, azaz ha változás történik a gráfban, az elemzést újra végre kell hajtani.

3. Használt fogalmak

Ebben a fejezetben az elemzés leírásánál használt fogalmakat tárgyaljuk részletesen, hogy az Erlangban kevésbé jártas olvasó is könnyedén átláthassa az algoritmus részleteit.

3.1. Folyamatok indítása

Erlang nyelven írt programok esetén az `erlang` modul `spawn/1,2,3,4`, `spawn_link/1,2,3,4` és `spawn_monitor/1,3` függvényeinek segítségével indíthatunk újabb folyamatokat. A különböző változatok segítségével megadható, melyik Erlang node-on induljon a folyamat, illetve melyik függvényt szeretnénk a folyamatban végrehajtani. Ezeket a paraméterek akár futási időben is megadhatóak, így kellően nagy szabadságot ad, hogy dinamikusan kezelhetőek legyenek.

Ez a szabadság megnehezíti a statikus elemzést, hiszen csak a forráskódban fellelhető információkkal tudunk dolgozni.

3.2. Folyamatok azonosítója

A folyamatok indításának visszatérési eredménye egy egyedi folyamat azonosító (`pid`). A `pid` segítségével a processzek globálisan címezhetőek, azaz ha több Erlang node van összekötve, akkor is garantált ezek egyedisége.

Egy folyamat a saját azonosítóját a `self()` függvény segítségével határozhatja meg.

3.3. Folyamatok regisztrálása

A folyamatok nevesítése a `register/2` függvény segítségével lehetséges. A regisztrált név csak egy Erlang node-on belül érvényes. A folyamat a nevének felhasználásával címezhető, anélkül hogy a folyamat azonosítója ismert lenne.

A folyamatok nevének globális regisztrálásához a `global:register_name/2,3` függvények használhatók. Ebben az esetben a folyamat tetszőleges Erlang node-ról címezhető ezzel a névvel.

3.4. Kommunikációs primitívek

Két folyamat kommunikációjára az alábbi primitívek használhatóak:

- `Pid ! Msg` – A `!` operátor segítségével üzenetet (`Msg`) küldhetünk a `Pid` azonosítóval rendelkező folyamatnak. A `Pid` helyett használható a folyamat neve is, amennyiben az regisztrálva lett.

- `erlang:send*/2,3` – Hasonlóan a `!` operátorhoz, ezekkel a függvényekkel üzenetet küldhetünk egy másik folyamatnak.

- `receive` – A `receive` konstrukcióval egy üzenetet fogadhatunk az üzenetsorról. A konstrukció lehetőséget ad szelektív üzenet fogadásra (megfelelő minta segítségével) és mindaddig blokkolódik, amíg nem sikerül a mintáknak megfelelő üzenetet fogadnia.

3.5. *ets* táblák

Az *ets* az Erlang beépített adattárolója, amely nagy mennyiségű adat tárolására ad lehetőséget és az adatok elérése konstans idejű. Az *ets* táblák létrehozásával egy újabb folyamat indul, amely az adat tárolásáért felel. A tábla mindaddig elérhető, amíg azt közvetlenül nem töröljük, vagy a szülő folyamat be nem fejeződik. A táblák indításakor különböző opciók segítségével befolyásolható annak típusa, láthatósági köre, neve, stb.

4. Az elemzés algoritmusa

Az elemző algoritmus a RefactorErl keretrendszer részeként lett megvalósítva. Az elemzés a szemantikus programgráf bejárásával kiterjeszti a gráfot, majd az így létrejött gráfból előállítja a program statikus kommunikációs modelljét.

Az elemző folyamat négy főbb lépésre osztható fel, melyet az alábbiakban részletesen ismertetünk.

4.1. Folyamatok felderítése

Az első lépésben az algoritmus lokalizálja azon pontokat a gráfban, ahol új folyamatok kerülnek elindításra. Ezután a következő lépésekben meghatározza a lehető legtöbb információt a folyamatról, amely statikus elemzések segítségével elérhető. Ez a lépés a szemantikus programgráfot bővíti újabb szemantikus `pid` típusú csúcsokkal és gráf élekkel.

4.1.1. Folyamatok meghatározása

Az elemzés kezdeti lépéseként meg kell határozni, hogy mely függvények indulhatnak különálló folyamatként. Az elemzés során a legpontosabb eredményre törekszünk, amely statikus elemzéssel előállítható. Ezért, ha szükséges, akkor adatfolyam elemzési eredményeket is felhasználunk. A folyamatokhoz rendelt csúcspontok és a folyamatokat indító kifejezések közötti kapcsolatot a `spawn_def` címkéjű él adja.

4.1.2. Regisztrált folyamatok

A nyelv, illetve a virtuális gép lehetőséget biztosít, hogy a folyamatokat globális névvel lássuk el. A folyamat ezek után elérhető mind az azonosítója, mind pedig a regisztrált neve segítségével is. A regisztrálással lehetőség nyílik, hogy a folyamat az azonosítójának hiányában is elérhető csupán a nevének ismeretével.

Ahhoz, hogy minél pontosabb legyen az elemzés, szükséges meghatározni, hogy mely folyamatok kerültek regisztrálásra és milyen névvel regisztrálták őket. Az elemzéshez itt is felhasználjuk az adatfolyam elemzést, mert a regisztrált név érkezhet paraméterként, így az explicit módon nem feltétlen jelenik meg a regisztrálás végző kifejezésben. Ha a

név csak futási időben kerül megadásra, azaz ha statikusan ez nem határozható meg a forráskódból, akkor ezen információ hiányában csökken az elemzés pontossága.

A beazonosított kifejezések és a folyamatok a `reg_def` címkéjű éllel kerülnek összekötésre. A regisztráláshoz használt nevek pedig a folyamatot reprezentáló csúcs egyik argumentuma lesz.

4.1.3. Kommunikáció felderítése

Következő lépésben a felderített folyamatok közötti lehetséges kommunikációt határozzuk meg. Első lépésként beazonosítjuk a küldést és fogadást végző kifejezéseket a folyamatok által végrehajtott függvényekben, majd adatfolyam elemzéssel meghatározzuk, hogy mely folyamatok között történt az üzenetváltás. A küldő, illetve a fogadó kifejezések közötti kommunikációt a `flow` címkéjű él jelöli.

4.2. Újabb folyamatok felderítése

A második lépésben további folyamatokkal bővítjük a meglévő folyamatokat. Ez a lépés a szemantikus gráfot bővíti újabb csúcsokkal és élekkel.

4.2.1. Kifejezések felderítése

A szemantikus gráfban megkeressük azon üzenetet küldő, üzenetet fogadó kifejezéseket, amelyek egyetlen folyamat végrehajtási útjában sem szerepelnek. Ezek azon függvényekben találhatóak, amelyek nem külön folyamatként indulnak, vagy statikus elemzéssel nem határozható meg hogy folyamatként indulhatnak. Minden egyes függvényhez, amely ilyen kifejezést tartalmaz, egy újabb csúcsot adunk meg.

4.2.2. Kifejezések folyamatkörnyezete

Miután a fennmaradó kifejezésekhez is hozzárendeltük a megfelelő folyamatokat, újabb élekkel bővítjük a gráfot. Az `eval_in` címkéjű élek azt határozzák meg, hogy az adott kifejezés melyik folyamatban kerül kiértékelésre. Természetesen csak az elemzés szempontjából releváns kifejezések (folyamat indítás, üzenet küldés, regisztrálás, stb.) és a folyamat azonosítók közé kerülnek behúzásra ezek az élek.

4.3. Gráf előállítása

Az elemzés harmadik lépése egy különálló gráfot állít elő, amely magába foglalja a folyamatokat és a köztük lévő kapcsolatokat.

A gráf csúcsai:

- A szemantikus gráfban is megjelenő folyamatokat reprezentáló csúcsok.
- A szuperprocessz (SP) csúcs, amely a környezet/virtuális gép szerepét tölti be.

A szuperprocessz csúcsnak kitüntetett szerepe van, ez alá kerül bekötésre minden olyan folyamat, amelynek nincs szülő folyamata.

A gráf élei:

- `spawn_link` – Egy folyamatot elindító és az ez általa elindított folyamat között jelenik meg, amely a processzek közötti szülő-gyermek kapcsolatot írja le.
- `register` – A regisztrálást végrehajtó folyamat és a regisztrált folyamat között jelenik meg.
- `spawn_sp` – A szülő nélküli folyamatok és a virtuális gépet reprezentáló szuperprocessz között kerül behúzásra.
- `{send, Message}` – Az üzenetet küldő és az üzenetet fogadó folyamat között jelenik meg. A `Message` a küldött üzenetet jelenti, amennyiben ez statikusan kinyerhető a forráskódból.

Ez így előállított gráf képezi a folyamatok kommunikációs modelljének alapját, amely még további információkkal bővülhet az elemzés következő fázisában.

4.4. Rejtett függőségek/kommunikáció

Az elemzés negyedik lépése újabb információkkal bővíti a szemantikus gráfot, valamint a különálló gráfot, amelyet az előző (4.3) fejezetben mutattunk be.

Az elemzés ezen fázisa elemzi az `ets` modul által definiált táblák létrehozását, írását és olvasását. Ahogyan azt a fogalmak áttekintésében leírtuk, az `ets` egy tábla szerkezet, amely e különálló folyamatként

létezik. A táblát a futó folyamatok írni és olvasni is tudják, ha az megfelelő opciókkal lett létrehozva. Ezáltal egy új kommunikációs csatorna nyílik a folyamatok számára.

4.4.1. Táblák létrehozása

`ETS` táblákat az `ets:new/2` függvény alkalmazásával lehet létrehozni. Elemezve ezen alkalmazásokat, meg tudjuk határozni, hogy mely pontokon keletkeznek ilyen táblák. Minden létrehozott táblának egy új `ets_tab` típusú szemantikus csúcsot hozunk létre mindkét gráfban.

A szemantikus gráfban a következő új élek jelennek meg:

- `ets_tab` – a gyökércsúcsból az `ets_tab` típusú csúcsba vezető élek.
- `ets_def` – az `ets_tab` típusú szemantikus csúcsból kiinduló él, amely a definiálási helyét határozza meg.
- `ets_ref` – az `ets_tab` típusú szemantikus csúcsból kiinduló él, amely a hivatkozási helyeit határozza meg.

A kommunikációs gráf egy új éllel bővül. A táblát létrehozó folyamat és a táblát reprezentáló csúcsok között a `create` címkéjű él kerül behúzásra.

4.5. Tábla olvasások

Az `ETS` táblákat több különböző függvény segítségével lehet olvasni. A szemantikus gráfot bejárva meghatározhatjuk, hogy ezek a függvények hol vannak alkalmazva. Ha egy folyamatban végrehajtódik valamelyik olvasást szolgáló függvény, az azt jelenti, hogy kommunikáció történik a táblán keresztül. Adatfolyam elemzéssel megpróbáljuk kideríteni, hogy melyik táblából történik az olvasás. Ha ezt sikerül meghatározni, akkor ezt a szemantikus gráfban az utasítást végrehajtó folyamatot és a táblát reprezentáló csúcsok között `read` címkéjű csúcs jelöli. A kommunikációs gráfot szintén bővítjük egy `{read, Data}` címkéjű éllel, ahol a `Data` az olvasott elem kulcsát, vagy a kereséshez használt mintát tartalmazza.

4.6. Tábla írások

Az `ETS` táblák írásához több függvény is adott. A szemantikus gráfból lekérdezhetőek a függvények alkalmazásainak helye, illetve hogy melyik folyamatban kerül végrehajtásra a függvény. Ezen információk ismeretében az adatfolyam elemzés segítségével meghatározzuk, hogy melyik táblában történt az írás. Amennyiben ez meghatározható statikusan, akkor a szemantikus gráfot egy új `write` címkéjű éllel bővítjük, amely a folyamat és az tábla között kerül behúzásra.

5. Kommunikációs gráf előállítása és megjelenítése

5.1. Elemzés végrehajtása

A RefactorErl shelljéből a `refanal_proc:anal_proc()` függvény kiértékelésével állíthatjuk elő a kommunikációs gráfot. Az elemzés lefuttatása által kibővül a szemantikus programgráf az új élekkel és szemantikus csúcsokkal, illetve létrejön a kommunikációs gráf.

A kommunikációs gráf csúcsait és a köztük futó éleket egy `processes` nevű `ets` táblában tároljuk. A gráf ebben a formájában is megtekinthető a virtuális gép tábla megjelenítőjével (`tv:start()` parancs), de ez nagy gráfok esetén nehezen áttekinthető.

5.2. Gráf megjelenítése

A `dot` gráf leíró nyelv segítségével közvetlenül emberek számára is könnyen olvashatóvá tehető a kommunikációs gráf. Ezért lehetővé tettük a gráf *dot* formátumba való exportálását. Ez a formátum átalakítható grafikus formátumra is, erre több program is lehetőséget biztosít.

Az elemzés végrehajtása után a `refanal_proc:create_dot()` függvény segítségével előállíthatjuk a *dot* formátumú fájlt. A függvény kiértékelése létrehoz az eszköz `data` könyvtárában egy `processes.dot` fájl, amely tartalmazza a teljes kommunikációs gráfot.

Unix alapú rendszereken a

```
dot -Tpdf processes.dot -o processes.pdf
```

utasítással egy *pdf* formátumú dokumentummá alakítható a *dot* fájl. Ter-

mészetesen nem csak *pdf* állítható elő, hanem különböző formátumokba (*svg*, *ps*, *gif*, *png*, stb) konvertálható.

A forráskód a `https://plc.inf.elte.hu/erlang/repos/branches/process_com` helyről érhető el.

5.3. Példa

Az alábbiakban tekintsük az alábbi két kliens-szerver Erlang modult (1. és 2. ábrák) és a belőlük készített kommunikációs gráfot (3. ábra). A szerver modul elindítja a `job_server`-t, mely kliensek csatlakozására vár, majd a tőlük érkező kéréseket fogadja és dolgozza fel. Érdekesség, hogy a kliensekkel való kommunikáció, az eredmények visszaadása, egy `ets` táblán keresztül történik. A kliens folyamatok csatlakoznak a szerverhez és a beolvasott értékek alapján a szerver felé kéréseket intéznek.

A 3. ábra mutatja, melyek azok a kapcsolatok, amelyeket statikusan fel tudott ismerni a bemutatott algoritmus.

6. Összefoglalás

Programok megértésének támogatása a programozók mindennapjait nagyban megkönnyíti. Különösen igaz ez olyan szoftverekre, ahol az egyszerűbb nyelvi elemek mellett párhuzamosságot, konkurenciát, elosztottságot támogató elemek is jelen vannak.

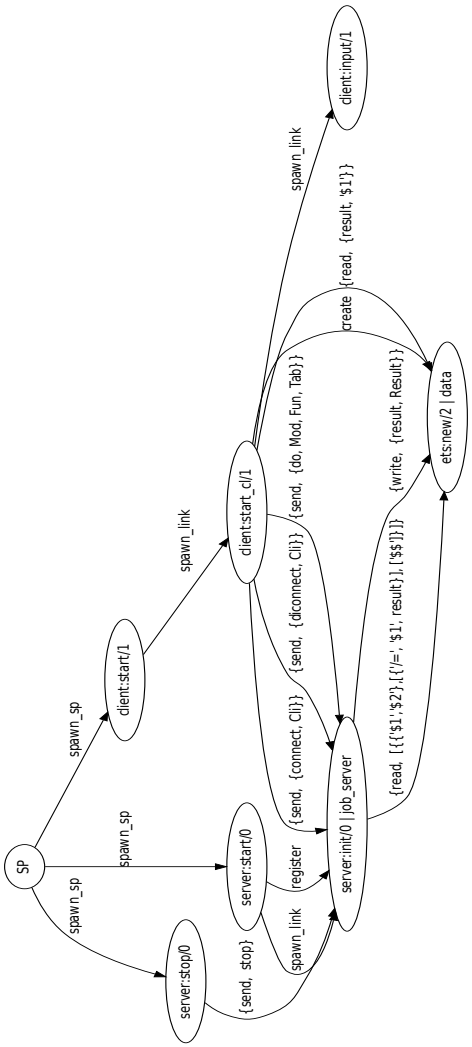
A RefactorErl elemző eszköz egyik célja, hogy a kódmegértést támogassa. Ebben a cikkben egy olyan kiterjesztését mutattuk be az eszköznek, mely segítségével statikusan elemzhető az Erlang folyamatok kommunikációja.

```
-module(server).

-export([start/0, stop/0]). %% Server interface
-export([init/0, loop/1]). %% Server callbacks

-define(Name, job_server).

%%%%%%%% Server interface %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start() -> register(?Name, spawn_link(?MODULE, init, [])).

stop() -> ?Name ! stop.

%%%%%%%% Server implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init()->
    process_flag(trap_exit, true),
    ?MODULE:loop([]).

loop(State)->
    receive
        stop -> ok;
        {connect, Cli} -> ?MODULE:loop([Cli|State]);
        {disconnect, Cli} ->
            ?MODULE:loop(lists:filter(fun(A) -> A /= Cli
                                  end, State));
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab,[{{'$1','$2'},[{'/=', '$1', result}],
                        ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).
```

1. ábra. Szerver modul

```
-module(client).
-export([start/1, input/1]).

-define(Name, job_server).

start(Client)->
    spawn_link(?MODULE, start_cl, [Client]).

start(Client) ->
    ?Name ! {connect, Client},
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        quit ->
            ?Name ! {disconnect, Name},
            io:format("~p~n", [ets:match(Tab, {result, '$1'})]);
        {job, {Mod, Fun}} ->
            ?Name ! {do, Mod, Fun, Tab},
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        quit ->
            Loop ! quit,
            ok;
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

2. ábra. Kliens modul



3. ábra. Folyamat kommunikációs gráf

Hivatkozások

[1] RefactorErl Home Page, 2011., `http://plc.inf.elte.hu/erlang/`

[2] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, M. Tóth, Refactorerl – source code analysis and refactoring in RefactorErl – Source Code Analysis and Refactoring in Erlang, *Proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia* 2011.

[3] F. Cesarini, S. Thompson, *Erlang Programming*, O'Reilly Media, 2009.

[4] Ericsson AB, *Erlang Reference Manual*, `http://www.erlang.org/doc/reference_manual/part_frame.html`

[5] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, M. Tóth, I. Bozó, R. Király, Modeling semantic knowledge in Erlang for refactoring, *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers, Presa Universitara Clujeana*, pp. 38–53.

[6] M. Tóth, I. Bozó, Static analysis of complex software systems implemented in Erlang, *Central European Functional Programming School*, LNCS 7241 (2012), Springer, pp. 440–498.

[7] M. Tóth, I. Bozó, Z. Horváth, M. Tejfel, First order flow analysis for Erlang, *Proceedings of the 8th Joint Conference on Mathematics and Computer Science, MaCS 2010.*