Statikus forráskódelemzés és -transzformálás

Habilitációs tézisek

2021.

Kozsik Tamás

http://kto.web.elte.hu/
kto@elte.hu

Eötvös Loránd Tudományegyetem
Informatikai Kar

Bevezető

Pályám kezdetén, már egyetemi tanulmányaim során felkeltette a figyelmem a párhuzamos és konkurens programozás problémaköre – az előbbivel TDK-dolgozatomban, az utóbbival diplomamunkámban foglalkoztam –, és ez az érdeklődés azóta is meghatározó eleme a kutatásaimnak. Az elmúlt mintegy 30 évben szerteágazó témákon dolgoztam, de a párhuzamos és konkurens programozásnak, valamint később az elosztott programok készítésének nehézségeit, illetve a nehézségek leküzdésének módjait újra és újra a kutatásom fókuszába kellett állítanom.

Vizsgáltam a bizonyíthatóan helyes programok fejlesztésének módszereit, különös hangsúllyal a konkurens és párhuzamos programokra. Terveztem alkalmazásiterület-specifikus nyelveket, köztük olyat, amely elosztott számítások ütemezéséhez használható. Statikus forráskódelemző és refaktoráló eszköz létrehozásában is közreműködtem – ezen a területen is a párhuzamos programozás támogatása volt a legfőbb eredményem.

Ebben a tézisfüzetben a Ph.D. fokozatom 2006-os megszerzése óta végzett tudományos munkámat mutatom be. Eredményeim a programozási nyelvektől és paradigmáktól a programozási módszereken és különböző programnyelvfeldolgozó eszközökön át a programok szematikájáig és helyességéig terjednek. Ezek közül a területek közül a statikus forráskódelemzés és -transzformálás emelkedik ki leginkább, így tézisfüzetemben erre helyezem a hangsúlyt. Ezen belül is kiemelem az oszd-meg-és-uralkodj elvű programok refaktorálással történő párhuzamosításának kérdését, melyet részletesebben tárgyalok. A harmadik fejezetben lazábban kapcsolódó eredményeimet foglalom össze igen tömören.

A szövegben előforduló számozott hivatkozások a doktori értekezésem utáni, az értekezésemben még nem szereplő saját publikációimat jelölik, például [22, 28]. Mások munkáira, valamint doktori értekezésem előtti munkáimra „[*szerző(k) évszám*]" formában hivatkozom, például [BD77].

Köszönetnyilvánítás

Mindig nagy örömet okozott az, ha egy kutatócsoportban, kollégáimmal, doktoranduszokkal, hallgatókkal együtt dolgozhattam. Eredményeink az együtt gondolkodás, az ötletelések, viták, kísérletezések során születtek. Nagyon köszönöm társszerzőimnek a közös munkát – amit ott és akkor talán nem is munkának, hanem szórakozásnak éreztünk.

Még inkább köszönöm családomnak, hogy a hosszú évek során elnézték nekem, hogy túl sok időt töltök a munkámmal, és túl keveset velük.

Köszönettel tartozom a Programozási Nyelvek és Fordítóprogramok Tanszék teljes közösségének az inspiráló közegért. Külön kiemelném Horváth Zoltán támogatását, aki TDK-munkám témavezetőjeként, oktatói–kutatói munkám elindítójaként, majd tanszékvezetőként és dékánként is mindent megtett szakmai fejlődésem érdekében.

A tézisfüzet elkészültét az Informatikai Kar belső pályázata segítette, valamint az Innovatív Informatikai és Infokommunikációs Megoldásokat Megalapozó Tematikus Kutatási Együttműködések című, EFOP-3.6.2-16-2017-00013 számú projekt, mely az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.

Tartalomjegyzék

Bevezető **ii**
Köszönetnyilvánítás . . . ii

1. Refaktoráló eszközök fejlesztése **1**
1.1. Elemző és refaktoráló infrastruktúra . . . 1
1.2. Változókötések elemzése . . . 3
1.3. Dinamikus nyelvi elemek elemzése . . . 4
1.3.1. Függvényhívások elemzése . . . 4
1.3.2. Függvények transzformálása esetleges kompenzációval . . . 5
1.3.3. Mellékhatások . . . 5
1.3.4. Forráskód felújítása . . . 6
1.3.5. Hatáselemzés . . . 6
1.4. Lexikális és szintaktikus információk . . . 6
1.4.1. Szintaktikus manipulációk . . . 7
1.4.2. Az előfeldolgozó problematikája . . . 7
1.4.3. A nyelvi fűszerek megőrzése . . . 8

2. Párhuzamosító átalakítások **9**
2.1. ParaPhrase Refactoring Tool for Erlang . . . 9
2.1.1. Illusztratív példa . . . 10
2.1.2. Mintafelismerés . . . 12
2.1.3. Minták rangsorolása . . . 13
2.1.4. Átalakítások . . . 15
2.2. *Oszd meg és uralkodj* algoritmusok párhuzamosítása . . . 15
2.2.1. Mintafelismerés . . . 17
2.2.2. Refaktorálás . . . 21

3. Egyéb eredmények **28**
3.1. Típusozás . . . 28
3.2. Erőforráselemzés és optimalizáció . . . 29
3.3. Alkalmazásiterület-specifikus nyelvek és speciális könyvtárak . . . 30
3.4. Programok szemantikája és helyessége . . . 31

Hivatkozott publikációim **33**

Irodalomjegyzék **37**

1. fejezet

Refaktoráló eszközök fejlesztése

2006-ban az egyik alapító tagja voltam annak a kutatócsoportnak, amely létrehozta a *RefactorErl*t, egy refaktoráló eszközt az Erlang nyelvhez. A kutatásvezető Horváth Zoltán volt, és velünk dolgozott több doktorandusz, illetve hallgató. A kutatásból négy doktori dolgozat született: Bozó István, Horpácsi Dániel, Király Roland, Tóth Melinda sikeresen doktoráltak. Sajnos többen nem (vagy még nem) készítették el doktori értekezésüket, de szerepük a projekt során kiemelkedő volt. Lövei László hozzájárulása az 1. fejezet alapjául szolgáló cikkekhez meghatározó volt. Jelentős volt (Nagyné) Víg Anikó és Nagy Tamás közreműködése az 1.1. fejezet eredményei elérésében. Meghatározó volt továbbá Kitlei Róbert és doktoranduszom, Poór Artúr munkája az 1.4. fejezetben bemutatott eredményekben.

A RefactorErl kutatócsoportnak 2006 és 2010 között voltam aktív tagja. **Fő feladatom az Erlang nyelv szabályainak és a szélsőséges eseteknek feltárása, a transzformációk vizsgálata, a transzformációk jelentést megőrző tulajdonságának elemzése, kompenzációs lehetőségek felderítése, a feltételrendszerek kidolgozása és rendszerbe foglalása, illetve cikkek megírása volt. Emellett részt vettem az elemző és refaktoráló szoftverinfrastruktúra tervezésében is** (a megvalósításában nem).

Számos eredmény és cikk elkészülésében volt szerepem; ezek közül kiemelendő a refaktoráló eszköz képességeit és használatának módját tárgyaló tanulmányunk [22], melynek első szerzője voltam. Ebben a tanulmányban bemutattuk egy elosztott programon, mint példán, hogy hogyan lehet, és miért érdemes programokat refaktorálni, illetve, hogy a RefactorErl eszközben akkor rendelkezésre álló 7 transzformáció (*Extract Function*, *Merge Expression Duplicates*, *Eliminate Variable*, *Rename Variable*, *Rename Function*, *Reorder Function Arguments*, *Tuple Function Arguments*) mire alkalmas. Nagyon részletesen kielemeztük ebben a cikkben azt is, hogy az egyes refaktoráló átalakításoknak milyen előfeltételeik vannak, és melyek azok a nyelvi elemek az Erlangban, amelyek nehézzé teszik akár az előfeltételek vizsgálatát, akár magát a transzformációt.

1.1. Elemző és refaktoráló infrastruktúra

Kutatócsoportunk versengve működött együtt a HaRe-t is kifejlesztő brit csoporttal. Az első jelentősebb publikációnkat [29] közösen készítettük, melyben már felhívtuk a figyelmet arra, hogy az Erlang programok refaktorálásánál az egyik legfontosabb hosszú távú célkitűzés a konkurens és elosztott programozást támogató nyelvi elemek elemzése, és az ezeket használó programok átalakítása lehet (kommunikációs minták, folyamatok struktúrája, defenzív versus let-it-crash programozás, OTP-sémák). A brit csoport a Wrangler nevű eszközt fejlesztette az Erlanghoz [LT08], mely szokványos módon, egy annotált absztrakt szintaxisfa (AAST) felépítésével és bejárásával működik. Ezzel szemben az általunk készített RefactorErl újszerű megközelítést alkalmaz: az úgynevezett *szemantikus programgráfot* (semantic program graph, SPG: szemantikus

információkkal kibővített AST) egy adatbázisban, perzisztensen tárolja. A relációs adatbázisban tárolás ötlete egyébként egy, a tanszékünkön korábban zajló kutatásnak volt az eredménye [DSNH04]. A szintaktikus és szemantikus információk külön táblákban való tárolását több cikkben is részleteztük [29, 34].

Korai publikációinkban az SPG-t egyszerűen csak „programgráf"-nak hívtuk, majd később [19] „szemantikus gráf"-nak. A szemantikus programgráf elnevezést 2010-ben javasoltuk [18].

A szemantikus programgráf egy olyan adatszerkezet, amely a programkód struktúráján (szintaktikus összefüggésein) kívül szemantikus kapcsolatokat és lexikális információkat is tartalmaz. Így három logikai rétegre bontjuk a refaktoráló eszköz rendelkezésére álló adatokat. Részletesebben tárgyaljuk ezt az 1.4. fejezetben, kitérve a lexikális információk szerepére is, de egyelőre elegendő azt látnunk, hogy az SPG-ben az absztrakt szintaxisfa csúcsai és élei mellett szemantikus csúcsok és élek is megjelennek.

Az SPG felépítésének és adatbázisban történő eltárolásának magas az idő- és tárhelyigénye, de ezek a költségek megtérülnek, ha ezt az adatbázist nem csak egy-egy refaktorálás végrehajtására használjuk, hanem refaktoráló lépések egy hosszabb sorozatát hajtjuk végre (ilyenre látunk majd példát a 2.2.fejezetben), mely lépések között nem, vagy csak részben kell újraelemezni a forráskódot. Hasonlóképpen hasznos a perzisztens tárolás, ha az elemzések során összegyűjtött információkat nem csak refaktorálásra használjuk, hanem például arra is, hogy kódmegértést elősegítő lekérdezéseket és a tárolt információk fölé épített további statikus forráskódelemzéseket futtassunk.

Az SPG perzisztálásának hasznát támasztotta alá azon módszertani javaslatunk [34], mely a forráskód fejlesztésének lépéseit a refaktoráló eszköz használatának szempontjából vizsgálja. Eszerint a refaktoráló eszközt úgy érdemes beépíteni az integrált fejlesztői környezetbe, hogy explicit módon meg lehessen különböztetni három különböző szerkesztési üzemmódot. A „kontrollált üzemmódban" csak jelentést megőrző transzformációkat hajthatunk végre, míg a „szabad szerkesztési üzemmódban" tetszőleges változtatások megengedettek. Amíg a kontrollált üzemmódban vagyunk, a kód újraelemzése nélkül tudjuk naprakészen tartani az SPG-t, így gazdaságosabb a perzisztens tárolás használata. Bármikor áttérhetünk a szabad szerkesztési üzemmódba, de amikor abból ismét a kontrollált üzemmódba kívánunk lépni, vagy üzemmódváltás nélkül egy refaktorálást végre akarunk hajtani, az SPG-t (vagy legalábbis azon forráskódmodulok részgráfjait és külső függőségeit, amelyek a szabad szerkesztési üzemmódban megváltoztak) újra kell építeni. A harmadik üzemmód a „korlátozott szerkesztési üzemmód". Ebben bizonyos szerkesztési feladatokat el lehet végezni, de csak olyanokat, amelyek hatása jól lokalizálható. Ilyen lehet például egy új függvény definiálása. Ennek az üzemmódnak az az előnye, hogy ezen szerkesztések során az SPG-nek egy kis részét kell csak újraépíteni, így az inkrementalitás biztosítható.

A refaktoráló eszközünk továbbfejlődése a fenti megközelítés elvetésével járt. Miután egyre több refaktoráló átalakítást valósítottunk meg, rájöttünk, hogy a szemantikus információk naprakészen tartása egy-egy transzformáció során túl nehéz feladat ahhoz képest, hogy mennyi költséggel jár ezek teljes újraszámolása. Ezért az eszközünk második generációjában egy új megoldással álltunk elő [12, 13]. A refaktoráló eszközben a transzformációk megvalósításai az SPG-ben csak a szintaktikus információk inkrementális naprakészen tartásáról gondoskodtak. A szemantikus csúcsokat és éleket minden transzformáció után újraépítette az eszköz.

A második generációs eszköz további nagy újdonsága az volt, hogy megpróbáltuk minél inkább szétválasztani az általános, nyelvfüggetlen elemző és transzformáló infrastruktúrát a nyelvspecifikus modelltől [12, 13]. A nyelvspecifikus modellt egy XML-dokumentumban, deklaratív módon adtuk meg, és az eszköz egyes komponensei ebből a modellből automatikusan lettek generálhatók. A nyelvspecifikus modell használata ellentmondott az abban az időben népszerű szemléletnek, mely egy nyelvfüggetlen refaktorálási metamodell létrehozását sürgette [TDDN00]. Cikkünkben [12] nyolc pontban érveltünk megközelítésünk mellett.

1.2. Változókötések elemzése

Az Erlang különböző nyelvei elemei különböző nehézségi szintet jelentenek egy statikus elemző számára, mely elemző végezheti például egy transzformáció feltételének ellenőrzését is. Vannak olyan nyelvi elemek, amelyek teljes mértékben kielemezhetők statikus elemzéssel, és természetesen vannak olyanok, melyek elvileg sem elemezhetők pontosan (l. Rice-tétel). A változókötések meghatározása [22, 32] az első kategóriába, míg például a függvényhívások feloldása [12, 22, 34] a második kategóriába tartozik.

Annak ellenére, hogy statikusan minden szükséges információt kinyerhetünk a forráskódból a változókról, a változókötések elemzése tartogat kihívásokat. A *változók* az Erlangban *nem változnak* ugyan, de érdekes, hogy egy változónak több helyen is adhatunk értéket (például egy elágazás különböző ágaiban), és hogy egy kifejezésről sokszor nem lehet pusztán ránézésre eldönteni, hogy az egy értékadás, azaz egy változókötés-e. Ennek az oka az, hogy az Erlang a szokványos nyelvektől eltérően kezeli a változókat.

A változók értéket mintaillesztés során kaphatnak. Ennek több formája is van: függvényhívás, elágazás, üzenetfogadás, kivételkezelő utasítás, egy listaalkotó kifejezés (list comprehension) generátora, vagy a fent látható mintaillesztő kifejezés. Furcsának tűnhet, de környezetfüggő, hogy egy mintaillesztés egy változónak értéket ad-e, vagy megvizsgálja a már meglévő értékét.

Az Erlang Reference Manual [BV99] leírja a változókötésekre vonatkozó szabályokat, de ezek az operációs szemantikára emlékeztető szabályok, amelyek a nyelvi konstrukciók input- és output-kontextusairól beszélnek, nem igazán kényelmesek a refaktoráló átalakítások definiálásához. Bevezettünk egy másik szabályrendszert, amellyel könnyebb megfogalmazni a változók *kötési struktúrájával* (binding structure) kapcsolatos feltételeket. Ehhez szükség volt olyan fogalmakra és definíciójukra, mint változók hatóköre és láthatósági köre, mintaillesztések hatóköre, hatókörhatároló, egy változó közvetlen előfordulása egy hatókörhatárolón belül, egy változó lokalitása, a kötő és nemkötő előfordulások mellett a potenciálisan kötő változóelőfordulások, valamint egy kifejezés egy másik kifejezéstől jobbra való elhelyezkedése [32]. Az általunk adott fogalmi rendszerben jól lehet kezelni a változók elfedését, a fordítóprogram-specifikus szabályokat (operandusok kiértékelési sorrendje), és azokat a specialitásokat (pl. ugyanannak a változónak a kötő és nemkötő előfordulásai egy adott mintában), amelyekre a refaktoráló transzformációk és előfeltételeik megfogalmazásához szükség van.

Felvetettünk [22] egy további (azóta is nyitott) problémát, amely heurisztikus megközelítést, illetve a programnyelvi szabályok mellett a stílus és az idiómáknak való megfelelés elemzését igényli. Az intelligens refaktorálás során a nyelvi szabályok értelmében különbözőnek tekintett változókat logikailag egységesen lehetne kezelni, ha a használt változónevek erre utalnak. Egy egyszerűbb példa erre az, amikor egy függvénydefiníció különböző klózaiban azonos pozíción álló formális paramétereknek ugyanazt a nevet adja a programozó; egy bonyolultabb példa az, amikor több, logikailag összetartozó függvényben egy adatra ugyanazzal a változónévvel utalunk. Az utóbbi rendszeresen előfordul az OTP-sémákkal definiált konkurens és elosztott programokban, amikor egy folyamat belső állapotát reprezentáljuk.

Egy intelligens refaktoráló eszköz számára megfogalmaztunk egy új szabályt, amely kompenzációs lehetőség felkínálását írja elő egy olyan esetre, amikor egy transzformáció elvégezhető a jelentés megőrzése mellett, de a kódminőség (pl. olvashatóság) romlását eredményezi [34]. Erre példa az, amikor egy változó átnevezése elfedést eredményez – ilyenkor egy intelligens refaktoráló eszköznek javasolnia kell, hogy az elfedéssel érintett másik változó is átnevezésre kerüljön.

Mindezeket a szabályokat figyelembe véve úgy alkottuk meg a változókkal kapcsolatos refaktorálásokat (változó átnevezése, egy kifejezés előfordulásainak változóba kiemelése, változó helyettesítése az értékével, kifejezéssorozat kiemelése egy új függvénybe), hogy a változókötések struktúrája megmaradjon.

1.3. Dinamikus nyelvi elemek elemzése

„Egy refaktoráló eszköz esetében alapvető fontosságú, hogy a programozási nyelv konstrukcióit minél jobban lefedjük statikus elemzéssel, és megállapítsuk, hogy mely feltételek esetén lehet egy transzformációt úgy elvégezni, hogy a kód jelentésének megőrzését garantálni lehessen” [33]. Minden furcsaságuk ellenére a változókötéseket tökéletesen ki lehet statikusan elemezni. Ugyanez már nem mondható el az Erlang sok más nyelvi eleméről. A bonyolultabb programtulajdonságok jól ismert *részleges eldönthetősége* a statikus elemzés és a refaktorálás alapvető korlátja minden Turing-teljes nyelvben. Az Erlangban rengeteg kihívást jelentő elem található, mint például a típusok dinamikus ellenőrzése (és így a változók polimorfizmusa), a magasabb rendű függvények, a reflektív programozás, a kifejezések mellékhatásai, a kivételek, valamint a konkurens és elosztott programozás egyes eszközei, mint az üzenetküldés.

Mivel a napi gyakorlatban előforduló programok jellemzően tartalmaznak erősen dinamikus nyelvi elemeket, nem tehetjük meg, hogy csak a statikusan teljesen elemezhető programokra koncentrálunk. Azt sem tehetjük meg, hogy csak azokat a transzformációkat engedjük végrehajtani a forráskódon, amelyek szemantikamegőrző tulajdonságát tökéletesen igazolni lehet, mert ezzel az eszközünk gyakorlati használhatóságát vetjük el. „Egy jó refaktoráló eszköz kellő mértékben biztonságos, de egyben nem túlságosan korlátozó kell legyen. Ennek érdekében a beépített döntéshozó mechanizmusok konfigurálhatók kell legyenek, vagy az eszköznek interaktívnak kell lennie” [12]. Az eszközt használó szoftverfejlesztő tapasztalatára, segítségére mindenképp szükség van ahhoz, hogy valódi programok elvárt szintű refaktorálását támogathassuk. Egy nagyon jó példa erre a problematikára a függvényhívások elemzése.

1.3.1. Függvényhívások elemzése

Az Erlangban vannak deklarált függvények, valamint explicit függvénykifejezések (lambda-függvények, mint pl. `fun (X) -> X+1 end`) és implicit függvénykifejezések (mint pl. `fun lists :map/2`). A nyelv magasabbrendű, így változókban is tárolhatunk függvényeket, illetve átadhatjuk őket paraméterként vagy függvények visszatérési értékeként. Önmagában már ez is szükségessé teszi azt, hogy adatfolyamelemzést végezzünk ahhoz, hogy megállapítsuk, hogy pl. egy paraméterként átadott függvény hol kerül meghívásra.

Még nehezebbé teszi a helyzetet, ha reflektív programozási módszereket használunk. Egy függvényt meghívhatunk úgy is, hogy a beépített `apply` függvénynek (vagy más hasonló függvénynek) átadjuk a meghívandó függvény nevét (mint adatot) és aktuális paraméterlistáját (szintén mint adatot). Így meghívható olyan függvény is, amelynek a nevét beolvastuk valahonnan, vagy üzenetben küldték nekünk. Tekintsünk egy példát [22], mely a *gen_server* OTP-séma leegyszerűsített változatának, egy elosztott szoftvernek lehet egy részlete. Itt egy `init/1` függvényt hívunk meg, de azt, hogy ez melyik modulban van definiálva, paraméterként kapjuk.

```
doStart(Name, Module, Args, ParentPid) ->
    {ok, State} = apply(Module,init,[Args]),
    register(Name,self()),
    ParentPid!started,
    loop(State, Module).
```

Ha magasabbrendű függvényt függvénykifejezéssel hívnak meg, vagy reflektív módszert használó függvényt függvénynévvel, mint atommal (ez egyfajta adattípus az Erlangban), vagy implicit függvénykifejezéssel hívnak meg, akkor a meghívott függvényt van esélyünk felismerni egy adatfolyamelemzéssel [13]. Ha viszont a függvénynevet (és/vagy a tartalmazó modul nevét, vagy a függvény paramétereit) beolvassuk vagy kiszámítjuk, akkor teljesen reménytelen helyzet elé állítjuk a statikus elemzőt [22]. Ugyanígy kivezet a statikusan elemezhető körből az `erl_eval` (az Erlang Meta Interpreter) használata is.

Refaktorálás során az elemzésnek azt is fel kell tudnia ismerni, amikor egy függvény neve úgy fordul elő a programban, hogy nem a függvényt azonosítja; egy függvényátnevezés során figyelmeztetést adhatunk erre az eszközünk használójának [34].

1.3.2. Függvények transzformálása esetleges kompenzációval

Az eddigiekből látható, hogy egy jó refaktoráló eszköznek le kell fednie a programozási nyelv szintaxisát, a szemantikából is minél nagyobb szeletet, és a szabványos programkönyvtárat is ismernie kell [34]. Ez teszi lehetővé, hogy a refaktoráló átalakítások például az `apply`-szerű függvényeken ne vérezzenek el. Egy függvény hívási helyeinek felismerése képezi az alapját sok refaktorálásnak.

Az átalakításoknál fel kell készülni a függvények különböző előfordulási formáira. A leggyakoribb eset, a közönséges hívás (a modulok közötti exportálási és importálási szabályokat is figyelembe véve) statikusan könnyen és biztosan felismerhető, az implicit függvénykifejezések szintén.

Az explicit függvénykifejezések transzformációja hasonlóképpen elvégezhető, és az adatfolyamelemzéssel megtalált hívási helyek az átalakítással összhangba hozhatók. Ha a statikus elemzés nem képes egzaktul megadni egy függvény hívási helyeit, kompenzáció alkalmazásával akkor is van lehetőség a szemantika megőrzésére a refaktorálás során. A kompenzáció, mely építhet mintaillesztésre vagy futási idejű típusellenőrzésre, alkalmazható a potenciális hívási helyszíneken, illetve az explicit függvénykifejezés körül is [22].

Hasonlóképpen alkalmazhatunk kompenzációt a reflektív hívások esetén is. Például egy `apply`-hívás helyszínén mintailleszthetünk egy konkrét függvényre, mely paramétereinek megváltozott a sorrendje [13]. Vagy ha átnevezzük az `m:init/1` függvényt `initialize`-ra, akkor a fentebb említett `doStart/4` függvényünket úgy alakíthatjuk át, hogy lecseréljük benne az `apply(Module,init,[Args])` hívást egy case-kifejezésre [22].

```
case Module of
    m -> m:initialize([Args]);
    _ -> apply(Module,init,[Args])
end
```

A kompenzációk csökkenthetik a kód olvashatóságát és hatékonyságát, ezért csak alapos megfontolás után érdemes ezt igénylő refaktorálásokat használni. Persze – ahogy már korábban is említettük a változók elfedésének kapcsán – van, hogy az eszköz épp az olvashatóság növelése érdekében ajánlhat kompenzációt. Egy adott nevű függvény átnevezésekor javasolhatja az eszköz, hogy az összes ugyanolyan nevű (más modulban, vagy más arítással definiált) függvény is konzisztensen átnevezésre kerüljön [34]. Például a fentebbi case-kifejezés bevezetésére nem lenne szükség, ha minden modulban átneveznénk az `init/1` függvényt `initialize`-ra.

1.3.3. Mellékhatások

Egy másik fontos elemzés, amelyre szükség lehet a refaktorálásoknál, a kifejezések mellékhatásainak feltárása. Ezek elsődleges forrása az Erlangban [PS11] az üzenetküldés és bizonyos beépített függvények használata (ide értve a klasszikus mellékhatás mellett azt is, amikor egy függvény sérti a hivatkozási helyfüggetlenséget, pl. `erlang:self/0`).[1] Mindezeket hasonlóan kell figyelembe vennünk a kifejezések mozgatásával járó átalakításoknál, mint a változókötést tartalmazó kifejezéseket. Figyelni kell arra, hogy az átalakítások során a mellékhatások száma és sorrendje ne változzon meg.

[1] Ide sorolhatnánk a különböző kivételeket is, de a refaktoráló eszköz gyakorlati használhatósága érdekében célszerű a kivételeket dinamikus szemantikai hibaként felfogni, és a mellékhatásoktól eltérő szabályok mentén kezelni [22].

1.3.4. Forráskód felújítása

Az adatfolyamelemzés nem csak a függvényhívások felderítéséhez használható, hanem például a programokban használt adatok szerkezetének megváltoztatásához is. Ennek egy jeles példája Erlangban az, amikor egy rendezett n-est rekorddá alakítunk. A rekordokat viszonylag későn vezették be az Erlang nyelvbe. Valójában ez a konstrukció egyfajta nyelvi fűszerként, szintaktikus cukorként jelent meg. Egy rekord nem más, mint egy címkézett rendezett n-es, de a nyelv fordítási idejű speciális extra lehetőségeket is biztosít hozzá (mint például mezők egy részének megváltoztatása a többi mező értékének meghagyása mellett). A forráskód felújításának (source code rejuvenation) nevezzük, amikor egy régebben írt kódot átalakítunk úgy, hogy az újabban bevezetett nyelvi elemeket is használja [PDS10].

A rekordok bevezetésével kapcsolatban mi is javasoltunk egy ilyen transzformációt [31]. Ebben az átalakításban kiindulunk egy rendezett n-esből, átalakítjuk rekorddá (bevezetve egy rekord deklarációt megadott rekord- és mezőnevekkel), majd iteratívan átalakítunk minden mintát, ahol a korábbi rendezett n-es illesztésre került, illetve minden kifejezést, amire egy átalakított mintát illeszt a kód. Az átalakítást az összefogó mintákon át is terjesztjük. Ha nem tud továbbterjedni az információ (pl. belefut egy módosíthatatlan forráskódú függvénybe), akkor explicit konverziós függvényhívást generálunk köré. Végezetül a rekordok módosításánál áttérünk a mezőfrissítő szintaxisra.

A forráskódfelújítás témakörével újabban a Java nyelv kapcsán foglalkozunk [1]. A nyelv 12-es verziójában bevezettek egy elegánsabb, biztonságosabban használható switch-utasítást, mellyel sok esetben helyettesíthető a hagyományos switch-utasítás – tapasztalataink szerint valós kódokban a hagyományos switch-utasítás előfordulásainak több, mint 90%-a. A hagyományos switch-utasításban minden olyan ág végére break-utasítást írunk, amelynél nem szeretnénk, hogy a vezérlés „átcsorogjon" a következő ágra. Az átcsorgás célzott használatával frappáns (bár a strukturált programozás elveit sértő) elágazási szerkezet hozható létre. Egy véletlenül kifelejtett break-utasítás viszont programhibát eredményez. Kidolgoztunk egy statikus elemzést a switch-utasítások kategorizálására, melyet a *PMD* statikus forráskódelemzőben[2] implementáltunk. A *JavaParser* eszközkészlet[3] segítségével megvalósítottuk azt a transzformációt is, amellyel egy régi switch-utasítás lecserélhető egy új switch-utasításra a szemantika megőrzése mellett.

1.3.5. Hatáselemzés

Visszatérve a RefactorErl projekthez, az adatfolyamelemzést kibővítettük hatáselemzéssé (lásd pl. [TuB+10], mellyel megállapíthatjuk, hogy egy adott kódrészlet változása befolyásolja-e egy másik kódrészlet működését [45]. Ennek számos haszna lehet. Például egy refaktorálás során alkalmazni szükséges kompenzációt csak ott kell elhelyeznünk, ahol a refaktorálás hatása kimutatható [13]. Kicsit szélesebb kontextusban vizsgálva ezt az eredményt, a hatáselemzéssel azt is eldönthetjük, hogy melyek azok a tesztesetek, amelyeket le kell futtatni egy programra egy változtatás (pl. egy refaktorálás) után: a nem releváns tesztesetek lefuttatását ezáltal megspórolhatjuk [45]. Ezen módszerek képezték az alapját két későbbi doktori értekezésnek [Boz18, Tót18].

1.4. Lexikális és szintaktikus információk

A lexikális információk kezelése ugyanolyan fajsúlyú probléma egy refaktoráló eszköz számára, mint akár a szintaktikus, akár a szemantikus információké. Szemben a fordítóprogramokba épített lexikális és szintaktikus elemzőkkel, egy refaktoráló eszköz front-endje nem hagyhatja veszni a megjegyzéseket, fehér szóközöket, vagy a redundáns zárójeleket: mindezekre szükség

[2] *PMD*: `https://pmd.github.io/`
[3] *JavaParser*: `https://javaparser.org/`

van, ha egy transzformáció után kinyomtatjuk a megváltoztatott forráskódot. Az eszköz használója ugyanis jogosan várhatja el, hogy a transzformációval nem érintett részeken a megjegyzések, a formázás és a lexikális információk megmaradjanak a forráskódban, sőt, hogy egy kódrészlet mozgatása is – amennyire lehet – megőrizze a mozgatott kód formázását.

1.4.1. Szintaktikus manipulációk

A RefactorErl első verzióját a de facto szabványnak tekinthető fordítóprogramban is használt lexikális és szintaktikus elemzővel készítettük el, mely nem adott a fenti problémákra kielégítő megoldást. A második verziónál viszont már megfogalmaztuk a „készítünk saját szintaktikus elemzőt" elv előnyeit [12]. Megközelítésünket egy XML-dokumentumban tárolt modellel támogattuk meg, mely tartalmazza az Erlang nyelv szintaktikus szabályait is. Ennek a modellnek nem csak az a szerepe, hogy a segítségével tudjuk szintaktikusan elemezni az eszközünk bemenetének adott forráskódot – a transzformációk megvalósításánál is ezt használjuk. A transzformációk a szemantikus programgráf vázát alkotó szintaxisfán végeznek el műveleteket: részfák törlése és mozgatása, illetve új részfák létrehozása és beillesztése. Mindezeket a nyelvspecifikus modell alapján, amennyire lehet, automatizáltan végzi az eszköz [19].

A szemantikus programgráf a következő összetevőkből áll [19].

- Egy gyökércsúccsal rendelkező irányított gráfból, melyben a csúcsok egy Erlang program nyelvi elemeit, az élek a közöttük lévő kapcsolatokat reprezentálják.
- A csúcsokhoz rendelt információkból – a csúcs *típusa* egy nyelvi fogalmat (*expr*, *variable*, *file* stb.) határoz meg, és a típustól függően további attribútumok tartozhatnak még hozzá (pl. egy bináris operátor alkalmazását jelentő kifejezésnél maga az alkalmazott operátor).
- Az élekhez rendelt címkékből, melyek a csúcsok közötti kapcsolat jellegét reprezentálják (pl. *sub* egy kifejezés részkifejezéseit mutatja).
- Az egy csúcsból kiinduló élek felett definiált rendezésből, amely meghatározza az elemek sorrendjét a programban (pl. különbséget tesz egy bináris operátor első és második operandusa között).

Az SPG-ben vannak lexikális, szintaktikus és szemantikus csúcsok, illetve élek. A szintaktikus csúcsok és élek egy AST-t határoznak meg, mely valóban elég absztrakt ahhoz, hogy könnyű legyen bejárni, feldolgozni. A konkrét szintaxis elemeit a lexikális csúcsokban tároljuk, így tudunk különbséget tenni az `(A+B)-C` és az `A+B-C` kifejezések között. A lexikális tokenek pozícióit is eltárolva gondoskodhatunk a forráskód formázásának megőrzéséről.

1.4.2. Az előfeldolgozó problematikája

Komoly kihívások elé állítja a refaktoráló eszközt, ha a vizsgált programozási nyelvhez előfeldolgozó (preprocesszor) is társul [Gar05, VBF04]. Az Erlang előfeldolgozó fájlbetöltések (include), makrók és feltételes fordítás leírására ad lehetőséget. Ipari célú kódban mindhárom lehetőség gyakran előfordul, így a RefactorErlt fel kellett készíteni az előfeldolgozás okozta problémákra [18].

Az alapvető megközelítés az, hogy a refaktoráló eszköznek az előfeldolgozás előtti állapotot is ismernie kell (hiszen ennek egy módosított változatát állítja elő és adja vissza egy-egy transzformáció után), de az előfeldolgozás utáni állapotra is szüksége van (hiszen ez az az állapot, amelyet elemezni kell, és amelynek a szemantikus állandóságát biztosítani kell). Az SPG-t tehát úgy kell kialakítani, hogy mindkét állapot reprezentációja helyet kapjon benne. Tárolni kell például a makrók definícióját, a makrók hívását, és a makrók kifejtéséből származó kódot is (ez utóbbi az AST-hez is hozzájárul), mindezt úgy, hogy kapcsolatuk az SPG-ből kiolvasható legyen.

A helyzetet tovább bonyolítja, hogy az Erlang előfeldolgozó támogatja a szintaxist keresztbevágó makrókat is[4]. Lehet, hogy a makró törzse egy önmagában szintaktikusan értelmezhetetlen tokensorozat, és az is lehet, hogy a makró kifejtése eredményez egy olyan tokensorozatot, amely a kifejtés helyén nem alkotja a szintaxisfa egy részfájának a frontját.

Még nehezebb feladat elé állítja a refaktoráló eszköz szerzőjét a feltételes fordítás, amelynek köszönhetően különböző fordítási beállítások esetén különböző előfordított forráskód állhat elő, melyeket mind reprezentálni kell az SPG-ben. Cikkünkben [18] megvizsgáltunk sok esetet, beleértve azt is, amikor a különböző előfeldolgozási lehetőségek összejátszanak egymással – és az igazán különleges esetek támogatásától el kellett tekintenünk.

1.4.3. A nyelvi fűszerek megőrzése

Egy másik kutatásban a Scala nyelvhez kezdtünk egy statikus elemző és refaktoráló szoftvert készíteni. Szerettük volna felhasználni a Scala fordítóprogramjában használt elemzések eredményeit – melyek könnyen hozzáférhetők, hiszen a Scalac infrastruktúra programozói interfészt biztosít az annotált absztrakt szintaxisfa eléréséhez. Ez a szintaxisfa azonban egy szintaktikusan leegyszerűsített kódot reprezentál, egy olyat, amelyből a nyelvi fűszerek (szintaktikus cukor, syntactic sugar) már eltávolításra került. Például egy `Array[String]` típusú `xs` változó esetén az `xs(2) = "two"` kifejezés egy fűszerezett változata az `xs.update(2, "two")` kifejezésnek – a szintaxisfa az utóbbi formát fogja tükrözni.

Azok a nyelvek, amelyek nagyon gazdag szintaxissal bírnak, gyakran egy magnyelv (core language) segítségével kerülnek ábrázolásra a fordítás során. Ez egyszerűsíti a fordítóprogram készítését, de az ezt előállító szintaktikus elemző nem lesz ideális egy refaktoráló eszközhöz.

A Scala nyelv kapcsán egy olyan megoldást javasoltunk [38], amely két elemző futásának eredményét fésüli össze. A Scala nyelvhez létezik egy olyan elemző is, a Scalameta[5], amely megőrzi a konkrét szintaxist, beleértve a nyelvi fűszereket is. Miután elvégezzük a Scala nyelvű forráskód szintaktikus elemzését a Scalametával, valamint a mélyebb elemzését a Scalac-vel, rendelkezésünkre áll egy gazdag szintaktikus információt tartalmazó AST, illetve egy leegyszerűsített szintaxist tartalmazó, de szemantikus információkkal (pl. típus) feldúsított AAST (annotált absztrakt szintaxisfa). Adtunk egy algoritmust, amellyel a gazdag szintaxist tartalmazó AST-be átmásoljuk az AAST-ből kinyert statikus szemantikai információt. A szintaxisfák egymásnak megfelelő csúcsait pozícióinformációk segítségével azonosítjuk. Sajnos az eljárás nem mindig ad tökéletes eredményt, mert a Scalameta és a Scalac helyenként nem „pozíció-konzisztens” (nem pontosan ugyanazt a pozícióinformációt tárolja egymásnak megfelelő terminális, illetve nemterminális szimbólumhoz). Megközelítésünkben az is problémát jelent, amikor a nyelvi fűszer eltávolítása (az ún. *desugaring*) megváltoztatja egy kifejezés típusát. Ezen anomáliák kiküszöböléséhez az eredmény utófeldolgozására van szükség – melynek megvalósítása további kutatásokat igényel.

[4]Szemben pl. a Lisp makróival, amelyek nem hághatják át a szintaxis korlátait [WC93].

[5]Scalameta. Library to read, analyze, transform and generate Scala programs. `http://scalameta.org`

2. fejezet

Párhuzamosító átalakítások

2013-ban az ELTE és az ELTE-Soft Kft. csatlakozott az FP7-es ParaPhrase projekthez, melynek keretén belül a *RefactorErl*re építve szekvenciális Erlang programok párhuzamossá alakításának módszereit, eszközkészletét fejlesztettük. Az ELTE részéről én, az ELTE-Soft részéről Horváth Zoltán volt a kutatásvezető (principal investigator). A RefactorErl projektben dolgozó doktoranduszokat és hallgatókat vontuk még be a kutatásba, mely jelentősen hozzájárult Bozó István, Horpácsi Dániel és Tóth Melinda megvédett doktori értekezéséhez. A további résztvevők, (Horpácsiné) Kőszegi Judit és Fördős Viktória munkája is meghatározó volt az eredmények elérésében – az ő esetükben a jövőben várható még fokozatszerzés.

Eredményeink ebben a kutatásban is nehezen bonthatók le személyekre. **Jómagam részt vettem a pályázat megírása, a projekt megtervezése és szakmai vezetése, valamint a publikációk elkészítése mellett az általános módszerek kifejlesztésében és az eszközök megtervezésében** – a módszerek részleteinek kidolgozása és az eszközök implementációja inkább a doktori hallgatók feladata volt. **Legjelentősebb szerepem az oszd-meg-és-uralkodj minta felismerésének és refaktorálással történő párhuzamosításának részletes kidolgozásában volt**, melyről két elsőszerzős folyóiratcikkem született [28, 27] (két nemzetközi konferencián történő előadás után), valamint meghívott előadóként is bemutattam egy hazai és egy külföldi nemzetközi tudományos konferencián [26, 21]. Emiatt ezt a témát kiemelten tárgyalom a 2.2. fejezetben. A projektben elért eredményeinket a szakma hazai képviselőinek is bemutattam a Szoftvertechnológiai Fórumon [20], illetve ismeretterjesztő cikket is készítettünk róluk [25].

2.1. ParaPhrase Refactoring Tool for Erlang

Dacára a több, mint fél évszázada zajló kutatásoknak és a jelentős előrelépésnek, a párhuzamos számítási erőforrások kihasználásának javítása még napjainkban is egy fontos kutatási terület. A ParaPhrase projekt [HAB+13] célkitűzése az volt, hogy módszereket és eszközöket adjon a szoftverfejlesztőknek arra, hogy heterogén (CPU-magokat és GPU-kat tartalmazó) architektúrákra gyorsabban, kényelmesebben, és főleg biztonságosabban tudjanak programokat készíteni. A projekt kidolgozott egy módszertant magas szintű párhuzamos programozási sémák [Col91, Col04] alkalmazására: a kódban ezekkel a sémákkal lehet leírni a párhuzamosságot, a sémákat pedig rá lehet képezni egy-egy adott (akár heterogén architektúrájú) gépre – sőt, akár dinamikusan meg is lehet változtatni a leképezést a rendelkezésre álló hardvererőforrások függvényében –, így téve lehetővé a könnyen megérthető és karbantartható, mégis hatékony párhuzamos szoftverek előállítását.

A módszertan egy fő elve az, hogy nem a programozónak kell a párhuzamos konstrukciókat (pl. a párhuzamos sémákat [GVL10, Loo12, ZHH06]) beírni a kódba, nem is a fordítóprogramra bízzuk rá a párhuzamosítást (akár a párhuzamos sémák beillesztésével [MIK97]), hanem a szof-

verfejlesztő egy megfelelő refaktoráló eszközt [MFM09, DME09, Dig11, BLH12] használ arra, hogy a számításokban a párhuzamosítható részeket felfedje, és refaktorálással átalakítsa ezeket a részeket egy párhuzamos programozási sémának (parallel pattern) megfelelően. Ez a megközelítés a kóddal való kísérletezést is támogatja: a refaktoráló eszközzel ugyanazt a kódot gyorsan átalakíthatjuk egyik vagy másik séma használatával párhuzamossá, és eldönthetjük, hogy melyik megoldás felel meg jobban a céljainknak. Így a ParaPhrase módszertan a párhuzamosító programtranszformációk szemi-automatizálásán alapszik.

A módszertan alátámasztására két programozási nyelvhez került sor eszközfejlesztésre: az egyik a C++, a másik pedig az Erlang. Az utóbbit nem igazán használják a nagy számítási teljesítményt igénylő (high-perfomance computing) területen, de megfelelőképpen komplementáris a C++ nyelvvel ahhoz, hogy érdekes választás legyen, valamint konstrukcióival támogatja a párhuzamosság alacsonyszintű megvalósítását [AS13]. Az Erlang programokban használható párhuzamos sémák a *Skel* könyvtárban lettek megvalósítva [BDH+14], és elkészült a heterogén architektúrás megvalósítás is, a *Lapedo* könyvtár [JBH15].[1]

A RefactorErl tudása azonban még tágabbra nyitotta a lehetőségeket. Segítségével létre tudtuk hozni a PaRTE (ejtsd: *parti*, azaz ParaPhrase Refactoring Tool for Erlang) nevű eszközt[2]. A PaRTE képes arra, hogy Erlang programokban automatikusan megkeressen jól párhuzamosítható részeket, ezeket rangsorolva bemutassa a szoftverfejlesztőnek, és el is végezze a párhuzamossá alakítást automatikus, illetve szemi-automatikus módon, úgy, hogy a *Skel* sémáit megfelelően felparaméterezve beilleszti a kiválasztott rész helyébe.[3]

2.1.1. Illusztratív példa

Cikkeinkben [6, 5, 28, 27, 46] példák segítségével elemezzük ki a ParaPhrase módszertan és a PaRTE eszköz használatát. Tekintsük itt most azt az `mxv/2` függvényt [5], amellyel ritkán kitöltött mátrixok és vektorok közötti szorzást definiálunk. Egy ritkát kitöltött oszlopvektort a nemnulla értékeiből és pozíciójukból álló párok listájaként (pozíciók szerint rendezve), egy ritkán kitöltött mátrixot pedig a legalább egy nemnulla elemet tartalmazó ritkán kitöltött sorvektoraik és pozíciójuk párjaiból álló listaként ábrázoljuk (ismét a pozíciók szerint rendezve).

```
mxv( Rows, Col ) ->
  DotProducts = [ {I,vxv(Row,Col)} || {I,Row} <- Rows ],
  lists:filter( fun({_I,V}) -> V /= 0 end, DotProducts ).
```

Az Erlang beépített eszközeivel párhuzamosíthatjuk `DotProducts` kiszámolását úgy, hogy minden skaláris szorzat (`vxv/2`) számításához új folyamatot hozunk létre [46].

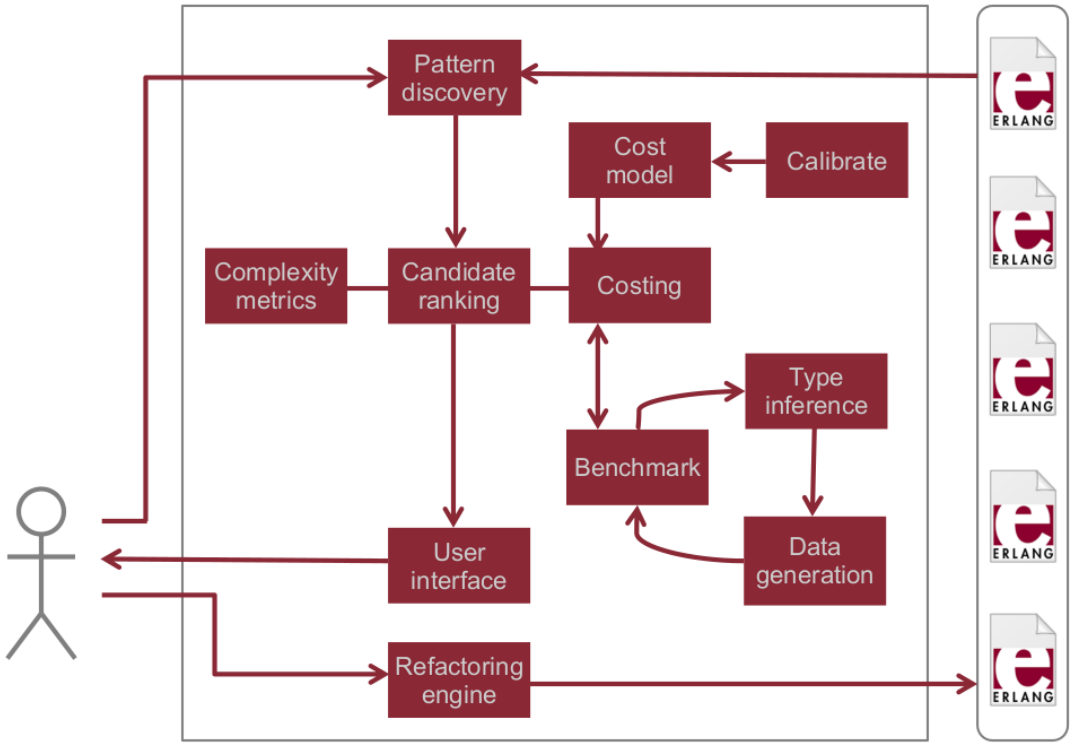
```
mxv( Rows, Col ) ->
  IndicesAndPids =
    [ {I, spawn(?MODULE, vxv_proc, [self(), Row, Col])}
      || {I,Row} <- Rows ],
  DotProducts =
    [ receive {Pid, Res} -> {I, Res} end
      || {I, Pid} <- IndicesAndPids ],
  lists:filter( fun({_,V}) -> V /= 0 end, DotProducts ).

vxv_proc(ParentPid, Row, Col) ->
  ParentPid ! {self(), vxv(Row, Col)}.
```

[1] Az Erlang lehetőségeire építve a D-Cleanhez [ZHH06] hasonló elosztott implementációjú sémák irányába is kibővíthető lenne a *Skel*.

[2] Wiki page of the ParaPhrase Refactoring Tool for Erlang, `http://pnyf.inf.elte.hu/trac/refactorerl/wiki/parte`

[3] A *Lapedo* heterogén architektúrára tervezett sémáival a projekt végéig nem sikerült összeépíteni az eszközünket. Ez még további fejlesztéseket igényel.



2.1. ábra. A PaRTE használata

Ez az átalakítás nem triviális, könnyű benne hibát véteni – és a kapott kód nehezen olvasható és karbantartható. Gondoljunk csak arra a szerencsétlen eshetőségre, hogy egy jószándékú fejlesztő alkalmazza az *Eliminate Variable* transzformációt (az 1. fejezetből) az `IndicesAndPids` és a `DotProducts` változókra, és ezzel visszacsinálja a párhuzamosítást [46]! A ParaPhrase megközelítés az *ad hoc* megoldások helyett a magas absztrakciós szintű sémák alkalmazását javasolja, ráadásul úgy, hogy ezeket a sémákat szoftvereszközzel lehessen bevezetni és manipulálni. Ennek megfelelően így alakítanánk át a mátrix-vektor szorzásunkat.

```
mxv( Rows, Col ) ->
  Fn = fun ({I,Row}) -> {I,vxv(Row,Col)} end,
  Computation = [{ord, [{farm,[{seq,Fn}],length(Rows)}]}],
  DotProducts = skel:do( Computation, Rows ),
  lists:filter( fun({_I,V}) -> V /= 0 end, DotProducts ).
```

A *Skel* könyvtár `skel:do/2` függvényének átadjuk paraméterként az elvégzendő számítás struktúráját leíró sémát (`Computation`) és a bemenő adatait. Az összetett séma leírásában egy szekvenciális komponens (`seq`) köré egy taszkfarmot (`farm`), aköré pedig az eredményeket rendezetten összegyűjtő komponenst (`ord`) teszünk. A módszertan további sémákat is támogat, mint például a *pipeline*-t (`pipe`), amellyel megfogalmazhatunk többlépcsős számolást – például egy három lépcsős pipeline kialakítható úgy, hogy a CPU-magokon végrehajtott előfeldolgozás után a következő lépcsőben GPU-kernelekkel dolgozzuk fel az adatokat, majd ismét a CPU-n végezzük az utófeldolgozást.

Mint látható, a megközelítés egy fontos eleme a kompozicionalitás. A sémák egymásba ágyazásával összetett sémákat, úgynevezett *workflow*-kat hozhatunk létre. A különböző workflow-k között szemantikus ekvivalenciák és átírási szabályok fogalmazhatók meg, melyek mentén megadatik a lehetőség a kódunk párhuzamosítási lehetőségei kapcsán a kísérletezésre [ACD98, BDH+14].

A PaRTE eszközzel a fenti átalakítás elvégezhető, de ennél valójában sokkal több szolgáltatást nyújt ez az eszköz. A 2.1 ábra bemutatja az eszköz használatának módját, illetve főbb összetevőit. A felhasználó a párhuzamosítani kívánt kódbázist betölti (létrehozza a szemantikus programgráfját), majd megkeresteti az eszközzel a kód párhuzamosítható részeit. A megtalált különböző párhuzamosítási mintákat az eszköz rangsorolva mutatja be a felhasználónak. A felhasználó kiválasztja, hogy melyik mintát kívánja refaktorálni, és milyen javasolt workflow-t kíván a kódba beilleszteni. A kért transzformációt az eszköz végrehajtja. Tekintsük át ezt részleteiben!

2.1.2. Mintafelismerés

A párhuzamosítható minták felismerése statikus forráskódelemzéssel egy mélyen kutatott téma mind a párhuzamosító fordítóprogramok, mind a refaktoráló eszközök kapcsán. A megoldás lényege, hogy az adat- [Tót18] és vezérlésifolyam [Boz18] tekintetében független részeket (*komponenseket*) kell azonosítani a számításban. Egy funkcionális nyelvben, mint az Erlang, ahol a mellékhatások kevésbé játszanak szerepet, mint egy imperatív nyelvben, ilyen független komponensek könnyebben találhatók. A mellékhatások vonatkozásában azt vizsgáltuk, hogy az egyes komponensek mellékhatásai függetlenek-e egymástól (*higiénikus komponensek*) [36, 6, 46].

Az elemzésben az is egy fontos szempont, hogy az azonosított kódrészletek elegendően nagyméretű számítást írjanak le, hogy megérje egyáltalán a párhuzamosítás. A mi szempontunkból még az is egy döntő tényező volt, hogy a ParaPhrase módszertanban rendelkezésre álló párhuzamos sémáknak (pl. `farm` és `pipe`) megfeleltethető mintákat azonosítsuk.

A mintafelismerés három területre összpontosult [6, Tót18]. Közülük a legkönnyebb a listákat elemenként feldolgozó szabványos programkönyvtárbeli függvények hívásainak megtalálása, leginkább a `lists` modulból (például `map`, `foldl` stb.), de más modulokból is (például `ordsets` és `orddict`, melyek listával valósítanak meg más adatszerkezeteket). Azok a műveletek is érdekesek, amelyek a könnyen listává alakítható adatszerkezeteken értelmezettek (`array`, `sets` és `dicts`), hiszen ezeknél egy kompenzáció (típuskonverzió) közbeiktatásával elvégezhető az alapvetően listákra kidolgozott párhuzamosítás. Ezen a területen a *RefactorErl*be beépített híváselemzés jól használható.

A következő érdekes terület a listageneráló kifejezések (list comprehension) elemzése. Ehhez már a szemantikus programgráfban tárolt szintaxisfa vizsgálatára van szükség. Azok a listageneráló kifejezések, amelyek fejében egy függvényalkalmazás található, potenciális taszkfarmként azonosíthatók – feltételezhetjük, hogy a meghívott függvény már eléggé komplex számítást valósít meg ahhoz, hogy megérhesse a párhuzamosítás. Ha ez a függvény a törzsében újabb függvényt hív meg, vagy ha a listageneráló kifejezés fejében több függvény hívását komponáljuk, akkor a minta egyben potenciális pipeline is lesz.

A harmadik terület a legizgalmasabb, mert bonyolultabb, szemantikus elemzés szükségeltetik hozzá: azonosítjuk azokat a rekurzív függvényeket, amelyek elemenkénti feldolgozást vagy előállítást végeznek (összefoglalóan *map-like* függvényeknek neveztük el ezeket). Itt olyan kódrészleteket találunk meg, amelyeket meg lehetett volna írni a `lists` modul magasabbrendű függvényeivel vagy egy listageneráló kifejezéssel, de a programozó ad hoc megoldást választva a rekurzív sémát összemosta az elemi számítások elvégzésével. Ezt a gondolatot továbbvittük, és kidolgoztuk a listákon értelmezett végrekurzív map-like függvények, a foreach-like függvények, valamint az adatfolyamot feldolgozó függvények karakterizálásának szabályait, továbbá a *pool evolution* minta [ACD+16] felismerésének szabályait is [46]. (Ezen utóbbi mintában egy evolúciós függvényt alkalmazunk egy adathalmaz elemeire úgy, hogy az eredmény visszakerül(het) az adathalmazba, és ezt iteratívan folytatjuk egy terminálási feltétel eléréséig.) A pontos szabályokat itt nem ismertetjük (lásd pl. [6, 46, Tót18]), de hasonló mintafelismerést részletesebben tárgyalunk a 2.2. fejezetben.

A mintafelismerés hatékonyságát megvizsgáltuk nyílt forráskódú Erlang szofverek programszövegén, és mindhárom elemzési területről számos mintát sikerült gyűjteni. A PaRTE a megta-

lált mintákat egy „mintaböngészőben” mutatja meg. Egy konkrét példát [6] mutat erre a 2.2. ábra a 14. oldalon. Ezen az ábrán már azt is láthatjuk, hogy az eszköz rangsorolja a talált mintákat.

2.1.3. Minták rangsorolása

A megtalált párhuzamosítható minták rangsorolása azért fontos, mert ez segít a szoftverfejlesztőnek abban, hogy hol érdemes elkezdenie a párhuzamosítást – melyik átalakításnak van a legnagyobb elvárt pozitív hatása. Két rangsorolási módszert dolgoztunk ki, melyek megőrzik azt a tulajdonságot, hogy a PaRTE eszközön belül teljesen automatikusan végrehajthatók.

Az első rangsorolási módszert a 11. oldal 2.1. ábráján a *Costing* elem és kapcsolatai reprezentálják [6]. Ennek a módszernek az alapötlete igen izgalmas: próbáljuk meg a háttérben lefuttatni a megtalált mintát, és mérjük meg, mennyire tűnik számításigényesnek. Technikailag rengeteg kihívást rejt ez a megközelítés. Először is egy statikus elemző eszközbe beépítünk egy dinamikus elemzést végző komponenst. Másodszor, ez a komponens nem az egész alkalmazást futtatja le, hanem csak egy kis részét. Ennek megfelelően a kiválasztott kódrészlet köré olyan körítést kell tudni építenünk, hogy a kapott kód a háttérben lefordítható és lefuttatható legyen. Harmadszor, ennek a generált kódnak megfelelő bemeneti adatokat is kell biztosítanunk, amelyekkel a futtatás elvégezhető, és a végrehajtási időből következtetések levonhatók.

Az első kihívást az Erlang/OTP rendszer eszközeivel kezelni lehet: egy programon belülről el lehet indítani egy fordítást és egy futtatást. A második kihívás is kezelhető egy programtranszformációkra szakosodott eszközben. A harmadik kihívás azt igényli, hogy megfelelő típusú, értelmezhető tesztadatot generáljunk egy statikusan elemzett kódrészlet számára. Tekintve, hogy az Erlang nyelv dinamikusan típusozott, már annak az eldöntése sem triviális, hogy egyáltalán milyen típusúak a vizsgált minta szabad változói (ezekből lesznek a generált program bemenetei, azaz ezek számára kell tesztadatot létrehozni). Mindenesetre, ha sikerül jó típusokat adni a függvényekhez, akkor a QuickCheck eszközzel [AHJW06] már tudunk megfelelő típusú tesztadatokat generálni. Ezt a megközelítést láthatjuk a 2.1. ábrán a *Benchmark*, a *Type inference* és *Data generation* komponensekben.

A *Benchmark* komponens tehát megméri, hogy egy mintajelölt (pattern candidate) egy mesterségesen generált bemenetre mennyi ideig fut – ezt az értéket még viszonyítani kell a párhuzamosítás becsült költségéhez, hogy megjósolhassuk, megéri-e a párhuzamosítás. A *Costing* feladata tehát egy potenciális gyorsulás (speedup) becslése. Ehhez a becsléshez egy költségmodellt (*Cost model*) építhetünk be a rendszerbe (pl. [BDH$^+$14]), amelyet kalibrálni kell a futtató számítógép konkrét adataival. A kalibrációhoz (*Calibration*) megmérjük a gép néhány paraméterét (folyamatok indításának és leállításának költsége, kommunikáció időigénye stb.).

Térjünk most vissza a típusozásra. Erlang programok statikus típusozására több módszer is született [AW93, AWL94, MW97, Nys03, LS06]. Mivel az Erlang programozók gyakran építenek az Erlang nyelv laza típusozási lehetőségeire, a más funkcionális nyelvekben megszokott típusozás az Erlang esetén nem igazán működik, inkább az altípusos rendszerek képesek a gyakorlatban előforduló programokat kezelni. A PaRTE-ban mi a TypEr [LS06] használata mellett döntöttünk. Az ebben alkalmazott *success typing* nem olyan típusozás, amely a „biztosan jól típusozott” tulajdonságot próbálja típuskikövetkeztetéssel bizonyítani, hanem azt tudja előrejelezni, ha egy függvény „biztosan rosszul típusozott”, azaz garantált, hogy futási hibát okoz. Ez az Erlang fejlesztők között népszerű módszer a gyakorlatban előforduló programokra is igen jól működik – a mi számunkra viszont nem volt ideális, mert túl általános típusokat adott meg. Nekünk egy köztes megoldás a megfelelő! Legyen a módszer alkalmas a gyakorlatban előforduló helyes programok típusának értelmezésére, másrészt viszont ne akarja a lehető legáltalánosabb típust meghatározni egyes kifejezésekhez: következtessen ki olyan típusokat, amilyen típusú adatot biztosan adhatunk egy függvénynek anélkül, hogy az elszállna típushibával. Erre a célra megkezdtük egy saját megoldás kidolgozását [35].

Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Expected speedup (GPU)	Recommended?
1	(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99	1,00	✓
2	(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80	1,00	✓
6	(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58	1,00	✓
3	(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58	1,00	✓
5	(Δe292)	matrix_ex_paper	mult_matrix2	2	12	2,98	1,00	✓
4	(Δe337)	matrix_ex_paper	scalar_product	2	12	1,06	1,00	✓

Chart options

Apply selected transformations

Details of the transformation sequence

Configuration	Location information	Program text	Number of workers	Sequential CPU time	Sequential GPU time	Parallel CPU time	Parallel GPU time	Expected speedup (CPU)	Expected speedup (GPU)	Used stream length
e337	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,15},{18,25}} - {{18, 30}, {18, 30}}	mult_scalar(A,B)	1	0,14	0,00	0,14	0,00	1,00	1,00	1
(Δe337)	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,13},{18,13}} - {{19, 39}, {19, 39}}	[mult_scalar(A,B) || {A,B} <- lists:zip(R,C)]	1	1 375,42	0,00	2 506,26	0,00	0,55	1,00	10 000
(Δ(Δe337))	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{6,3},{6,3}} - {{7, 26}, {7, 26}}	[scalar_product(R,C) || R <- Rows, C <- Cols]	12	13 754 154,08	0,00	2 091 407,67	0,00	6,58	1,00	10 000

Chart options

2.2. ábra. Párhuzamosítható minták egy mátrixszorzásos kódban

A mintajelöltek benchmarkolásán alapuló módszer elméletben nagyon izgalmas, de a gyakorlatban sajnos elég korlátozott a használhatósága. Az adattípusok kikövetkeztetése Erlangban nehéz, és a típusinvariánsok meghatározására egyáltalán nem dolgoztunk ki módszert – a típusinvariánsok megértése nélkül viszont egyáltalán nem lehet értelmes tesztadatokat generálni. A ritka mátrixok és ritka vektorok szorzását megvalósító függvény például a listák pozíció szerinti rendezettségére épít. Ha ezt nem teljesíti a generált tesztadat, a művelet nem viselkedik ésszerűen. Vannak esetek, amikor pusztán az adattípusok elegendőek a tesztadatok létrehozásához, mint például a (nem ritkán kitöltött) mátrixok szorzása esetén, amit a 2.2. ábrán is láthatunk.

Mindenesetre szükség volt egy alternatív rangsorolási módszer kidolgozására is, és ehhez egy kódmetrika alapú megközelítést választottunk. Megmértük a mintajelöltben található számítás kódjának (Erlanghoz igazított) méretét (az AST csomópontok számát), mégpedig úgy, hogy a számítás során meghívott függvények komplexitását is tranzitíven hozzászámítottuk, valamint a rekurzív hívások esetén egy jelentős tényezőt hozzávettünk (*Complexity metrics*).

2.1.4. Átalakítások

Miután a *Pattern discovery* befejeződik, és a refaktorálásra érdemes mintákat rangsorolva felkínálja a *Pattern candidate browser*, a PaRTE felhasználója kiválaszthat egy javasolt *workflow*-t, és végrehajthatja a refaktoráló eszközzel a párhuzamosítást. A refaktorálás három lépésben történik [6, 5]. Először egy kanonikus alakra hozzuk a mintát (shaping), majd lecseréljük a megfelelő összetett sémára és a *Skel* könyvtár használatára. Végezetül rendrakó (clean-up) transzformációkat hajtunk végre a kódon. A több egyszerű lépésre bontott refaktorálás gondoskodik arról, hogy az egyes lépések érthetőek, esetleg bizonyíthatóan helyesek maradjanak.

Egy `farm` bevezetéséhez kanonikus alaknak például egy olyan listageneráló kifejezést választottunk [6], amelynek csak egy generátora van, nincs szűrője, és a fejében egy unáris függvény alkalmazása áll. A párhuzamosító átalakítás az `mxm/2` fenti kódján egy ilyen kanonikus alakon keresztül vezeti be a taszkfarm-sémát, majd automatikusan alkalmaz rendrakó, kódcsinosító átalakításokat is.

A refaktorálások megadásához bevezettünk egy alkalmazásiterület-specifikus nyelvet, melyben a szintaktikus átalakítások metaváltozókat tartalmazó Erlang progamrészletekkel fogalmazhatók meg, míg a feltételek és egyéb összefüggések szemantikus lekérdezésekkel írhatók le [5]. Ezek az eredmények és a refaktorálások helyességének vizsgálata megjelentek Horpácsi Dániel és Kőszegi Judit munkáiban [HKT16], valamint Horpácsi Dániel doktori értekezésében [Hor18].

2.2. *Oszd meg és uralkodj* algoritmusok párhuzamosítása

A vizsgált párhuzamosító átalakítások közül a legizgalmasabb – és legnehezebb – az oszt-meg-és-uralkodj (divide-and-conquer, röviden *d&c*) elvű algoritmusok felismerése és refaktorálással történő párhuzamosítása. Ezek az algoritmusok a természetüknél fogva jól párhuzamosíthatók, és mivel nagyon sok alkalmazási területen találkozhatunk velük, érdemes egy párhuzamos sémát bevezetni az általános kezelésükhöz.

A megközelítésünk az eddigiekhez hasonló lesz: definiáljuk azokat a statikus elemzéseket, amelyekkel fel lehet ismerni ezeket az algoritmusokat [28], majd a metrika alapú bonyolultságbecslési módszerrel rangsorolni lehet a talált mintajelölteket, és végül megkezdhetjük a kiválasztott minta refaktorálását, mely előkészítő lépésekből (shaping) és a séma bevezetéséből (valamint esetleges rendrakásból / clean-up) áll. A refaktorálás módszertana azonban kissé eltér a korábbiaktól: nem automatizáltuk az előkészítő átalakítások végrehajtását, csak egy forgatókönyvet adtunk ahhoz, hogy hogyan lehet az átalakításokat szemi-automatizáltan végrehajtani [27].

Egy *d&c* számítás a megoldandó feladatot felbontja kisebb részekre, és rekurzívan megoldja a kisebb részproblémákat. Ezt a rekurzív felbontást addig folytatja, amíg a megoldandó probléma

el nem ér egy „alapesetet”, melyet további felbontás nélkül is képes már megoldani. A *d&c*-viselkedést illusztrálja az alábbi (funkcionális paradigmát követő) pszeudokód. Ebben a `solve` rekurzív hívásáért a részfeladatokon a közismert **`map`** magasabbrendű függvény felel.

```
solve(Problem) =
    if is_base_case(Problem)
        then solve_base_case(Problem)
        else SubProblems = divide(Problem)
             Solutions = map(solve, SubProblems)
             combine(Solutions)
    end
```

Mou és Hudak [MH88] *divacon*nak hívja azokat az f függvényeket, melyek viselkedése valamilyen felbontható („non-basic”) adatokon felírhatók ebben az alakban.

$$c \circ h \circ (\text{map } f) \circ g \circ d$$

Itt d egy felbontó, c pedig egy komponáló függvény, míg g és h ún. „igazító”-függvények. Érdekes módon a *divacon* függvények közé tartozik például a `map` és a `fold` is, sőt – Mou és Hudak épp ezt hozza fel példaként [MH88]. A mi szemszögünkből nézve ezek azonban degenerált esetek, melyeket mi nem kívánunk *d&c* függvényként kezelni. Számunkra a párhuzamos sémák bevezetése a fő cél, és ezeket a szélsőséges eseteket más párhuzamos mintaként (pl. `farm` és `reduce`) kívánjuk felismerni.

Váltsunk nézőpontot, és fókuszáljunk arra, hogy ez az absztrakt szerkezet egy adott programozási nyelven leprogramozva igen változatos alakokat ölthet, amelyekben nem feltétlenül könnyű megtalálni ezt az elrejtett struktúrát. A célunk első lépésben az, hogy felismerjük és feltárjuk a struktúrát, a második lépésben pedig az, hogy explicitté tegyük egy magasabbrendű `dc` művelet megfelelően felparaméterezett hívásával.

```
solve = dc(is_base_case,solve_base_case,divide,combine)
dc(IsBase,Base,Divide,Combine) =
    let Solve(Problem) =
          if IsBase(Problem)
              then Base(Problem)
              else SubProblems = Divide(Problem)
                   Solutions = map(Solve, SubProblems)
                   Combine(Solutions)
          end
    in Solve
```

Természetesen a *d&c* szerkezet felismerése és egy magasabbrendű függvényhívással való explicitté tétele nem csak a párhuzamosítás szempontjából érdekes, hanem önmagában is. Koncentráljunk itt most azonban a párhuzamosításra, mely innen már csak egy lépésre van. Ha rendelkezésünkre áll a **`map`** párhuzamos változata, a **`parmap`** magasabbrendű függvény, akkor a párhuzamosított *d&c* az alábbi `pardc` művelet meghívásával fejezhető ki.

```
pardc(ShouldBeSequential,IsBase,Base,Divide,Combine) =
    let Solve(Problem) =
          if ShouldBeSequential(Problem)
              then dc(IsBase,Base,Divide,Combine)(Problem)
              else SubProblems = Divide(Problem)
                   Solutions = parmap(Solve, SubProblems)
                   Combine(Solutions)
          end
    in Solve
```

A gyakorlatban általában nem érdemes a *d&c* probléma megoldását teljes mélységben párhuzamosítani, valamilyen felső korlátot érdemes szabni a párhuzamos dekompozíciónak. Ez a korlát (`ShouldBeSequential` a példánkban) hivatkozhat az elindított folyamatok számára, vagy a részproblémák méretére, bonyolultságára. Gazdag lehetőséget ad a párhuzamosság korlátozására például az Eden sémakönyvtár [Loo12] *d&c*-implementációja is, hiszen a túlzott párhuzamosítás a hatékonyság rovására mehet. Mások is foglalkoztak a *d&c* séma hatékony párhuzamos megvalósításával [Her01], sőt, azzal is, hogy refaktorálással párhuzamosítsanak ezen séma bevezetésére építve: Freisleben és Kielmann [FK95] azonban nem használ erre mintafelismerést, helyette a programozónak kell annotációkat elhelyezni a forráskódban a megfelelő helyekre. Mintafelismerést támogat ellenben a SkelML [MIK97], mely egy párhuzamos sémákat használó fordítóprogram. A SkelML fold-előfordulásokat képes *d&c*-vé alakítani, viszont – fordítóprogramról lévén szó – a mintafelismerés ultrakonzervatív, sokkal kevesebb mintát ismer fel és alakít át, mint a mi megközelítésünk.

2.2.1. Mintafelismerés

A *d&c* függvényeket a következőképpen karakterizáljuk: *olyan függvény, amely többször is meghívja saját magát egymástól független adatokon ugyanazon a végrehajtási úton.*

Az általunk használt statikus elemzés alapvetően szintaxis-alapú, a mintakeresés az elemzett program szintaxisfájából indul ki. Így fontos a különböző nyelvi, szintaktikus lehetőségek áttekintése, melyeket konkrét példákon tanulmányozunk. Ezután a megfigyelések alapján megadjuk a mintafelismeréshez használt szabályokat.

A *d&c* előfordulások szintaktikus gazdagsága

Tekintsük először a Merge-sort alábbi implementációját.

```
ms(  [] ) ->  [];
ms( [H] ) -> [H];
ms(  L )  ->
    {L1, L2} = lists:split( (length(L) div 2), L ),
    merge( ms(L1), ms(L2) ).
```

Ezen a példán megfigyelhetjük a fenti feltételek teljesülését: az `ms/1` függvényben létezik egy végrehajtási út, amelyen meghívja magát kétszer is, mégpedig úgy, hogy a második híváshoz használt paraméter nem függ az első hívás eredményétől.

A feltételek akkor is teljesülnek, ha egészen más szintaktikus alakba írjuk a definíciót. A függvényklózok helyett használhatunk case-kifejezést, a két explicit hívást elrejthetjük egy `lists:map/2` hívásába, és a végrehajtási út is kiterjedhet több függvénydefinícióra (inter-procedurális vezérlésifolyam-elemzést igényelve). A szükséges paraméterek is több mintaillesztésen át juthatnak el a felhasználási helyükre, ezt adatfolyamelemzéssel követhetjük le.

```
ms(L) ->
  case L of
    [] -> [];
    [H] -> [H];
    _ -> sort_and_merge(lists:split((length(L) div 2), L))
  end.

sort_and_merge( {L1, L2} ) ->
  [Sorted1, Sorted2] = lists:map( fun ms/1, [L1,L2] ),
  merge( Sorted1, Sorted2 ).
```

A szintaktikus lehetőségek mellett a szabványos programkönyvtár speciális kezelése is fontos, mint ahogy a `lists:map/2` példáján látjuk. Sok *d&c* algoritmusnál a rekurzív hívások száma

dinamikusan alakul, így ténylegesen a kanonikus alakra emlékeztető `lists:map/2`-hívás szerepel a kódban, mint a Radix-sort alábbi megvalósításában.

```
rs(  [],    _   ) ->  [];
rs( [V],    _   ) -> [V];
rs(List, Level) ->
  Buckets = divide(List,Level),
  SortedLists =
    lists:map( fun(B) -> rs(B,Level+1) end, Buckets ),
  lists:append(SortedLists).
```

A listageneráló kifejezések fejében szereplő rekurzív hívások is hasonló módon kezelendők; a *d&c* minta felismerendő például akkor is, ha a fenti kód harmadik függvényklózát lecseréljük egy listageneráló-kifejezéses megoldásra.

```
rs(List, Level) ->
  lists:append([rs(B,Level+1) || B<-divide(List,Level)]).
```

Az eddigi példák mind lista típusú paraméterrel dolgoztak, de a felismerő eljárásnak más adattípust is kezelnie kell tudnia – például nagy egész számokat reprezentáló bitsztringeket, mint az alábbi gyorsszorzásos példában.

```
karatsuba(Num1, Num2) ->
  S1 = bit_size(Num1),
  S2 = bit_size(Num2),
  case {Num1, Num2} of
    ...
    _ ->
      M = max(S1, S2),
      M2 = M - (M div 2),
      <<Low1:M2/bitstring, High1/bitstring>> = Num1,
      <<Low2:M2/bitstring, High2/bitstring>> = Num2,
      Z0 = karatsuba(Low1,Low2),
      Z1 = karatsuba(add(Low1,High1),add(Low2,High2)),
      Z2 = karatsuba(High1,High2),
      add(  add(shift(Z2, M2*2), Z0),  shift(sub(Z1,add(Z2, Z0)), M2)  )
  end.
```

A következő példa, a Minimax-eljárás alábbi kódja azt az esetet szemlélteti, amikor kölcsönösen egymást hívó (mutually recursive) függvényekben ismerjük fel a *d&c* mintát.

```
mm_max(Node, Depth) ->
  case Depth == 0 orelse terminal(Node) of
    true  -> value(Node);
    false -> lists:max([mm_min(C,Depth-1)||C <- children(Node)])
  end.
mm_min(Node, Depth) ->
  % hasonló, csak az mm_max-ot hívja mm_min helyett
  % és a lists:min-t a lists:max helyett
```

Egy függvény többszöri meghívását egymástól független adatokon elérhetjük egy *map-like* vagy egy *fold-like* függvény (lásd a 2.1.2. fejezetet) segítségével is. Egy *d&c* függvényben ez is hordozhatja a keresett struktúrát, mint az alábbi példákban.

```
rs(List, Level) ->
    lists:append(conquer(divide(List,Level),Level)).

conquer([],Level) -> [];    % map-like
conquer([B|Bs],Level) -> [rs(B,Level+1)|conquer(Bs,Level)].
```

```
rs(List, Level) ->
    conquer(divide(List,Level),Level).

conquer([],Level) -> [];    % fold-like
conquer([B|Bs],Level) -> rs(B,Level+1)++conquer(Bs,Level).
```

Még tovább általánosíthatjuk az előző két példát, és az alábbi kódszerkezethez jutunk, mely szintén *d&c* függvényt eredményez.

```
x(P) -> ...  r(fun x/1, partition(P)) ...

r(F,Q) ->
  A = some_part_of(Q),
  B = some_other_part_of(Q),
  ...
  C = F(A),     % A ne függjön D-től
  D = r(F,B),   % B ne függjön C-től
  ...
```

Ellenpéldák *d&c*-re

Még jobban megértjük a mintakeresést, ha meggondoljuk, hogy milyen esetekben *nem* tekintjük *d&c*-nek a függvényünket. Észrevehetjük, hogy a legegyszerűbb rekurzív függvények általános alakja is hasonlít a *d&c*-hez – de ezt nyilván nem akarjuk *d&c*-mintaként felismerni.

```
f(Problem) ->
  case base_case(Problem) of
    true  -> basic_function(Problem);
    false -> SubProblem = divide(Problem),
             SubSolved = f(SubProblem),
             combine(SubSolved)
  end.
```

A hasonlóság még inkább szembetűnő, ha az ötödik sorban szereplő kifejezést helyettesítjük a vele szemantikusan ekvivalens alábbival.

```
             [SubSolved] = lists:map(fun f/1, [SubProblem])
```

Ha fel tudjuk ismerni, hogy egy függvény nem hívja meg magát többször rekurzívan (itt: a `lists:map/2` második paramétere egy egyelemű lista), akkor el tudjuk kerülni azt, hogy *d&c*-ként ismerjük fel. Ugyanezt lehet mondani sok map-like és fold-like függvényről [28].

A következő ellenpélda a végrehajtási utak elemzésének fontosságára hívja fel a figyelmet. Szintaktikusan két rekurzív hívás is található az alábbi függvényben, de különböző végrehajtási utakon – így a függvény nem tekintendő *d&c*-nek.

```
binsearch(Array,Pattern) ->
    binsearch(Array,0,array:size(Array)-1,Pattern).
binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
    H = (Lower+Upper) div 2,
    Val = array:get(H,Array),
    if
      Val < Pattern -> binsearch(Array,H+1,Upper,Pattern);
      Val > Pattern -> binsearch(Array,Lower,H-1,Pattern);
      true          -> true
    end;
binsearch(_,_,_,_) -> false.
```

Az, hogy mennyire precíz a vezérlésifolyam-elemzés, amellyel a végrehajtási utakat tárjuk fel, visszahat arra, hogy az eszközünk különbséget tud-e tenni az alábbi két, szintaktikusan hasonló függvény között (az első nem *d&c*, de a második, a Quicksort funkcionális változata az).

```
binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
  H = (Lower+Upper) div 2,
  Val = array:get(H,Array),
  (Val == Pattern)
      orelse (Val<Pattern andalso binsearch(Array,H+1,Upper,Pattern))
      orelse (Val>Pattern andalso binsearch(Array,Lower,H-1,Pattern));
binsearch(_,_,_,_) -> false.

qs( [H|T] ) ->
    {List1, List2} = lists:partition(fun(X)-> X<H end, T),
    Left  = if length(List1) > 1 -> qs(List1);
               true              -> List1
            end,
    Right = if length(List2) > 1 -> qs(List2);
               true              -> List2
            end,
    Left ++ [H] ++ Right
  end.
```

Az utolsó példa pedig a Quicksort-szerű függvényünk átírata egy akkumulátoros, végrekurzív segédfüggvény bevezetésével. A számítás elve ugyanaz, mint az előbbi *d&c*-nek kategorizált példában, de a végrehajtási vermet most szimuláljuk a `qc/2` második paramétere segítségével. A vezérlés ilyen mérvű adattá konvertálása lehetetlenné teszi az elemzőnk számára a *d&c*-viselkedés észlelését és kiaknázását.

```
qs(List) -> lists:reverse(qs([],[List])).

qs(Result, [])               -> Result;
qs(Result, [[] | Lists])     -> qs(Result,Lists);
qs(Result, [[H] | Lists])    -> qs([H|Result],Lists);
qs(Result, [[H|T] | Lists]) ->
        {SubList1, SubList2} = lists:partition(fun(X)-> X < H end, T),
        qs(Result,[SubList1, [H], SubList2 | Lists]).
```

Felismerési szabályok

A *d&c*-felismerés szabályainak tárgyalása előtt említést kell tennünk néhány olyan elemzésről, amelyekre a szabályok építenek. Szükségünk van az Erlang nyelv specialitásaihoz igazított vezérlésifolyam- [Boz18] és adatfolyamelemzésekre [Tót18]. Az előbbi segítségével értelmezzük a végrehajtási utak (execution path, EP) fogalmát, és mindkettőre szükség van a függőségi reláció (dependence, $\overset{\text{dep}}{\rightsquigarrow}$) definiálásához. Az interprocedurális vezérlésifolyamgráfban a függvényhívások kezelésére speciális csúcsokat szokás bevezetni.

- $start_g$ reprezentálja a g függvény végrehajtásának kezdetét.
- end_g reprezentálja a g függvény végrehajtásának befejeződését.
- $call_g^c$ reprezentálja a g függvény meghívását a c kifejezésben.
- ret_g^c reprezentálja a g függvény visszatérését a c kifejezésben.

Ezen jelölések mellett a *d&c*-felismerési szabályok már értelmezhetők. A mintakeresés során megvizsgáljuk, hogy egy f függvény megfelel-e az alábbi felismerési szabályoknak. Kiindulunk az f-ből induló interprocedurális vezérlésifolyamgráfból és az abból számolható végrehajtási utakból, valamint az f-ből induló függőségi gráfból. Az f-et *d&c* mintaként ismerjük fel, ha az alábbi három feltételnek megfelel.

1. f rekurzív: létezik olyan végrehajtási útja, amelyen meghívja magát;
$$\exists p \in EP(start_f),\ \exists c \text{ amelyre } call_f^c \in p$$
2. f-nek van alapesete: létezik olyan végrehajtási útja, amelyen nem hívja meg magát;
$$\exists p \in EP(start_f) \text{ amelyre } (\nexists c : call_f^c \in p) \wedge (end_f \in p)$$
3. f több független rekurzív hívást tartalmaz egy végrehajtási útján, azaz az alábbi három eset valamelyike fennáll.
   - Létezik olyan végrehajtási útja, amelyen van legalább két független rekurzív hívása:
   $$\exists c_1, c_2, \exists p \in EP(ret_f^{c_1}) \text{ amelyre}$$
   $$call_f^{c_2} \in p\ \wedge\ \forall a \in \mathrm{ARG}(c_2) :\ \neg(a \overset{\mathbf{dep}}{\rightsquigarrow} ret_f^{c_1})$$
   ahol ARG a függvényhívás aktuális paramétereit reprezentáló AST-csúcsokat jelenti.
   - Létezik olyan végrehajtási útja, amely olyan listageneráló kifejezést tartalmaz, amelynek a h fejrésze közvetlenül vagy közvetve meghívja f-et:
   $$\exists p \in EP(h),\ \exists c \text{ amelyre } call_f^c \in p.$$
   - Közvetlenül vagy közvetve meghív egy „farm-jelölt" függvényt[4], g-t, amely meghívja f-et minden rekurzív végrehajtási útján.
   $$\exists p \in EP(start_f), \exists c_1, \exists g \text{ rekurzív függvény, amelyre } call_g^{c_1} \in p \wedge$$
   $$\forall q \in EP(start_g) : (\exists c_2 :\ call_g^{c_2} \in q) \rightarrow (\exists c_3 :\ call_f^{c_3} \in q)$$

Az ezen szabályok mentén megvalósított mintafelismerő elemzésünk a bemutatott pozitív példákat felismeri, a negatív példákat pedig elutasítja. Emellett megnéztünk több nyílt forráskódú Erlangban írt szoftvert, melyekben számos *d&c*-jelöltet találtunk, közöttük olyat is, amelyről „szabad szemmel" nem is látszott, hogy az.

2.2.2. Refaktorálás

A következő lépés a *d&c*-minták átalakítása a megfelelő párhuzamos séma bevezetésével [27]. Ehhez előkészítő refaktorálási lépéseket hajt végre a PaRTE használója, amíg el nem ér egy kanonikus alakot, melyből aztán a *d&c*-séma bevezethető. A *Skel* könyvtárban ez a séma másképp néz ki, mint az eddig tárgyaltak: a `sk_hlp:dc_lim/5` magasabbrendű függvényt kell meghívni (ahol a hlp a high-level pattern rövidítése).

Tekintsük át az előkészítő átalakításokat. A leírásokban írógép-betűtípust használunk az Erlang-nyelv szintaxisához, és dőlt betűkkel jelöljük a metaváltozókat: ε (kifejezés), p (minta), g (őrfeltétel), b (törzs, azaz kifejezéssorozat), f (függvényszimbólum), V (változó). Szintaktikus elemek (σ_i, ahol $i \in [1..n]$) az Erlang konkrét szintaxisában vesszővel vagy pontosvesszővel elválasztott sorozatát $\langle\sigma_i\rangle_{i\in[1..n]}$ jelöli. Az itt használt absztrakt szintaxisban a függvényklózok és case-kifejezések őrfeltételeit mindig kiírjuk (a konkrét szintaxisban a mindig igaz őrfeltétel elhagyható). Így például egy egyparaméteres, egyklózú függvény definícióját így adjuk meg: $f(p)$ `when` $g \rightarrow b$. A transzformációkat a $\Rightarrow$ szimbólum jelzi, illetve a $\Leftrightarrow$, ha mindkét irányú átalakítás hasznos lehet a céljainkra.

[4] A map-like függvények és különböző általánosításai tartoznak ebbe a kategóriába, lásd a 2.1.2. fejezetet.

Function Clauses to/from Case Clauses. Egy függvény klózait egybeolvaszthatjuk egyetlen klózzá úgy, hogy a klózokat meghatározó mintaillesztéseket bevisszük egy legkülső case-kifejezésbe. Ugyanez visszafelé is jól használható. Az egyszerűség kedvéért egyparaméteres függvényekre adjuk meg a szabályt. (A *Tuple Function Arguments* átalakítással egy többparaméteres függvényt könnyen egyparaméteressé tehetünk.

$$\langle f(p_i)\ \texttt{when}\ g_i \to b_i\rangle_{i\in[1..n]} \quad \overset{?}{\equiv} \quad f(V) \to \texttt{case}\ V\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i\rangle_{i\in[1..n]}\ \texttt{end}$$

Ezen átalakítás feltétele, hogy a V változó nem fordul elő a mintákban, az őrfeltételekben és a törzsekben. (Ugyanolyan nevű *másik* változó előfordulhat a törzsekben, például egy explicit függvénykifejezés formális paraméterlistájában.) A transzformáció a szemantikusan helyes programok jelentését mindkét irányban megőrzi, de a nyelv dinamikus szemantikai szabályainak megsértése (sikertelen mintaillesztés) miatt más kivételeket eredményez a két változat.

```
ms([])   ->  [];
ms([H]) -> [H].
```

⇔

```
ms(List) -> case List of [] -> [];
                         [H] -> [H]
            end.
```

Group Case Branches. Ezzel az átalakítással szétválaszthatjuk egy case-kifejezés ágait két csoportba. A *d&c*-átalakítások során így választhatjuk szét egy függvénydefiníció alapeseteit a rekurzív esetektől. Az átalakítás alkalmazásánál meg kell adni, hogy mely ágak tartoznak a **true** -, és melyek a **false**-csoportba. A választást itt most a $\beta : [1..n] \to \{\text{true}, \text{false}\}$ metafüggvény fejezi ki.

Az átalakítás egy case-kifejezésből négy másikat csinál. A p'_i minták és a g'_i őrfeltételek a p_i minták és a g_i őrfeltételek egyfajta másolatai: az ott kötött változók helyébe friss változókat vezetünk be. Azért, hogy elkerüljük a nem használt változók miatti fordítási figyelmeztetéseket, a friss változók neveit aláhúzásjellel kell kezdeni.

$$\begin{aligned}
&\texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i\rangle_{i\in[1..n]}\ \texttt{end}\\
\overset{?}{\equiv}\\
&\texttt{case}\ \Big(\texttt{case}\ \varepsilon\ \texttt{of}\ \langle p'_i\ \texttt{when}\ g'_i \to \beta(i)\rangle_{i\in[1..n]}\ \texttt{end}\Big)\ \texttt{of}\\
&\quad \texttt{true} \to \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i\rangle_{i\in[1..n]\wedge\beta(i)=\text{true}}\ \texttt{end};\\
&\quad \texttt{false} \to \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i\rangle_{i\in[1..n]\wedge\beta(i)=\text{false}}\ \texttt{end}\\
&\texttt{end}
\end{aligned}$$

Az átalakítás feltétele, hogy a kiindulási case-kifejezés fejében szereplő ε kifejezés tiszta legyen. (A minták és az őrfeltételek mindig tiszták Erlangban, ezért rájuk nem kell hasonló feltételt megfogalmazni.)

Az alábbi példában alkalmazzuk először ezt az átalakítást, majd a case-kifejezés fejében lévő újabb case-kifejezésre egy *Introduce Variable* transzformációt.

```
ms(L) ->
  case L of
    []  -> [];
    [H] -> [H];
    _   -> {L1,L2} = lists:split( length(L) div 2, L ),
           lists:merge( ms(L1), ms(L2) )
  end.
```

⇓

```
ms(L) ->
  IsBase = case L of
             []   -> true;
             [_H] -> true;
             _    -> false
           end,
  case IsBase of
    true  ->
        case L of
          []  -> [];
          [H] -> [H]
        end;
    false ->
        case L of
          _ -> {L1,L2} = lists:split( length(L) div 2, L ),
               lists:merge( ms(L1), ms(L2) )
        end
  end.
```

Eliminate Single Branch. Amikor egy case-kifejezés csak egyetlen ágat tartalmaz, egy mintaillesztő kifejezéssel kezdődő kifejezéssorozattá alakíthatjuk. Ehhez további feltételt nem is kell szabnunk. (Ha a mintaillesztés sikertelen, különböző kivételeket kapunk a két esetben, de egyébként a két kifejezés ekvivalens.)

$$\texttt{case}\ \varepsilon\ \texttt{of}\ p \to b\ \texttt{end} \quad \stackrel{?}{\equiv} \quad \texttt{begin}\ p = \varepsilon,\ b\ \texttt{end}$$

Ha a case-kifejezés nem egy másik kifejezés részkifejezése, akkor a begin-end pár is elhagyható. Ha statikusan bizonyítható, hogy a mintaillesztés mindig sikeres, és a mintaillesztés nem köt meg egy változót sem, akkor a mintaillesztő kifejezés is elhagyható a jobboldalról.

Ezt a rendrakó transzformációt alkalmazhatjuk az előző példa negyedik case-kifejezésére: csak az ág törzse marad meg belőle. Egy másik példa az általános esetet szemlélteti.

```
case L of [H] -> [H] end
```
⇒
```
begin [H] = L, [H] end
```

Introduce lists:map/2. A *d&c*-függvények többször is meghívják magukat rekurzívan. A kanonikus alakban ezt a `lists:map/2` hívásával fogjuk kifejezni. Ezt készíti elő ez a transzformáció: ha egy listakifejezésben több elem szerepel, és mindegyik elem ugyanannak az egyparaméteres függvénynek a hívásával áll elő, akkor a `lists:map/2` hívásával helyettesíthetjük a listakifejezést.

$$\left[\ \langle f(\varepsilon_i)\rangle_{i\in[1..n]}\ \right] \quad \stackrel{?}{\equiv} \quad \texttt{lists:map}\left(\texttt{fun}\ f\texttt{/1},\ \left[\ \langle \varepsilon_i\rangle_{i\in[1..n]}\ \right]\right)$$

A transzformáció feltétele, hogy vagy a ε_i kifejezések, vagy az f törzse legyen tiszta. (Különben a mellékhatások sorrendje megváltozna.)

```
[ms(L1), ms(L2)]
```
⇒
```
lists:map(fun ms/1, [L1,L2])
```

Hasonló átalakításról már a 2.1.4. fejezetben is volt szó: egy egygenerátoros listageneráló kifejezést lecserélhetünk egy `lists:map/2` hívásra [5].

Bindings To List. Az előző transzformációt előkészítendő, szükség lehet arra, hogy mintaillesztő kifejezések egy sorozatát egyetlen mintaillesztő kifejezéssé alakítja, amelyben listamintára illesztünk egy listakifejezést.

$$\langle p_i = \varepsilon_i \rangle_{i\in[1..n]} \quad \stackrel{?}{\equiv} \quad \left[\ \langle p_i \rangle_{i\in[1..n]}\ \right] = \left[\ \langle \varepsilon_i \rangle_{i\in[1..n]}\ \right]$$

Teljesen biztonságossá teszi ezt a transzformációt az a feltétel, hogy a ε_i kifejezések tiszták. Ha ez nem teljesül, és a programrészlet dinamikus szemantikai hibát tartalmaz (valamelyik p_j-re nem illeszkedik a megfelelő ε_j), akkor a lista-alapú mintaillesztés több mellékhatás végrehajtását eredményezheti a kivétel fellépése előtt, mint a mintaillesztések sorozata.

```
S1 = ms(L1), S2 = ms(L2)
```

$\Rightarrow$

```
[S1,S2] = [ms(L1), ms(L2)]
```

Reorder Expressions. Ezzel a transzformációval megváltoztathatjuk a legfelső szintű kifejezések sorrendjét egy törzsben (kifejezéssorozatban).

$$\varepsilon_1,\ \varepsilon_2 \quad \stackrel{?}{\equiv} \quad \varepsilon_2,\ \varepsilon_1$$

A transzformáció feltétele, hogy a második kifejezés ne használjon olyan változót, amelyet az első kötött meg (mert ez megváltoztatná a változók kötési struktúráját, ha egyáltalán értelmes maradna a program), és hogy legfeljebb az egyik kifejezés lehet csak mellékhatásos. (Ez gyengíthető, ha a mellékhatások függetlenek egymástól, azaz a sorrendjük tetszőleges.)

Move Expression Out Of Case. Amikor egy case-kifejezésben minden ág utolsó kifejezése pontosan ugyanaz, akkor ezt a kifejezést kivihetjük a case-kifejezés utánra.

$$\texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i,\ \varepsilon' \rangle_{i\in[1..n]}\ \texttt{end} \quad \stackrel{?}{\equiv} \quad \texttt{begin}\ \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i \rangle_{i\in[1..n]}\ \texttt{end},\ \varepsilon'\ \texttt{end}$$

A begin-end elhagyható, ha a case-kifejezés egy törzs legkülső szintű kifejezése.

```
case Problem of
  {{Node, Depth}, max} ->
    F = fun lists:max/1,
    D = min,
    work(Depth, Node, F, D);
  {{Node, Depth}, min} ->
    F = fun lists:min/1,
    D = max,
    work(Depth, Node, F, D)
end
```

$\Rightarrow$

```
case Problem of
  {{Node, Depth}, max} ->
    F = fun lists:max/1,
    D = min;
  {{Node, Depth}, min} ->
    F = fun lists:min/1,
    D = max
end,
work(Depth, Node, F, D)
```

Move Expression Into Case. Egy case-kifejezés előtt álló kifejezést bevihetünk a case-kifejezésbe, oly módon, hogy minden egyes ág elején megismételjük.

$$\varepsilon', \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to b_i \rangle_{i\in[1..n]}\ \texttt{end} \quad \stackrel{?}{\equiv} \quad \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \to \varepsilon',\ b_i \rangle_{i\in[1..n]}\ \texttt{end}$$

Ez az átalakítás megváltoztatja a kifejezések kiértékelési sorrendjét, ezért hasonló feltételt kell szabnunk hozzá, mint a *Reorder Expressions* transzformációhoz. Az ε' kifejezésben kötött változók nem fordulhatnak elő az ε kifejezésben, a p_i mintákban és g_i őrfeltételekben. Az ε' és ε kifejezések közül legalább az egyik legyen tiszta (vagy ha mégsem, akkor legyen a mellékhatásuk független egymástól).

Introduce Unused Parameter. A *d&c*-implementáció 3 összetevője, az `is_base_case/1`, `solve_base_case/1` és `divide/1` függvények ugyanolyan paramétert várnak (`Problem`). Ahhoz, hogy a három műveletet ugyanarra az állapottérre tudjuk hozni, szükség van egy olyan átalakításra, amely egy új, nem használt paramétert vesz fel egy függvénybe.

Az átalakítás kiegészíti a formális és aktuális paraméterlistákat. A formális paraméterlistában egy névtelen paraméter is megfelel, hiszen a paramétert nem használjuk a függvényben.

$$\Big\langle f(\langle p_{i,j}\rangle_{j\in[1..m]}) \texttt{ when } g_i \to b_i \Big\rangle_{i\in[1..n]} \quad \Rightarrow$$
$$\Big\langle f(\langle p_{i,j}\rangle_{j\in[1..m]}, \_) \texttt{ when } g_i \to b_i \Big\rangle_{i\in[1..n]}$$

Továbbá az f minden c hívására választunk egy ε_c kifejezést.

$$f(\langle \varepsilon_j\rangle_{j\in[1..m]}) \quad \Rightarrow \quad f(\langle \varepsilon_j\rangle_{j\in[1..m]},\ \varepsilon_c).$$

Az átalakítás feltétele, hogy az ε_c kifejezések tiszták legyenek, és statikusan bizonyítható legyen, hogy nem váltanak ki kivételt.

A transzformációt érdemes egy függvényhívásnál kezdeményezni egy új aktuális paraméter megadásával. A függvény további hívásait érdemes egy `undefined` atommal, mint új aktuális paraméterrel kiegészíteni.

Merge Function Definitions. Ez az átalakítás több függvénydefiníciót összegyúr egyetlen függvénydefinícióvá. Az új definíció klózai az eredeti függvénydefiníciók klózai lesznek. Az új függvénybe felveszünk egy új paramétert, ami a diszkriminátor szerepét tölti be, és az új függvény klózaiban mintaillesztünk erre a diszkriminátorra.

$$\langle f_j(p_{i,j}) \texttt{ when } g_{i,j} \to b_{i,j}\rangle_{i\in[1..n_j],j\in[1..m]} \quad \Rightarrow$$
$$\Big\langle f(p_{i,j},\beta(j)) \texttt{ when } g_{i,j} \to b_{i,j} \Big\rangle_{i\in[1..n_j],j\in[1..m]}$$

Az f_j függvények minden hívását is átalakítjuk.

$$f_j(\varepsilon) \quad \Rightarrow \quad f(\varepsilon,\ \beta(j)).$$

Az átalakítás feltétele, hogy az új függvény neve egy friss név legyen, β legyen injektív, és az f_j függvények vagy mind exportáltak legyenek, vagy egyikük se legyen exportált. Ha az eredeti függvények exportáltak voltak, akkor kompenzációként az új függvényt is exportálttá kell tenni.

Ezt a transzformációt használhatjuk például a Minimax-algoritmus (lásd a 18. oldalon) kanonikus alakra hozása során, mely egy ponton az alábbi alakot ölti [27].

```
mm( {Node, Depth}, max ) ->
  ...
    lists:max([mm({C,Depth-1},min)||C<-children(Node)])
  end;
mm( {Node, Depth}, min ) ->
  ...
    lists:min([mm({C,Depth-1},max)||C<-children(Node)])
  end.
```

A Merge-sort átalakítása

Az eddig bemutatott transzformációkkal a Merge-sort függvényünk az alábbi alakra hozható [27].

```
ms(Lst) ->
  case is_base_case(Lst) of
    true  -> solve_base_case(Lst);
    false -> SubProblems = divide(Lst),
             Solutions = lists:map(fun ms/1, SubProblems),
             combine(Solutions)
  end.

is_base_case( [] ) -> true;
is_base_case([_H]) -> true;
is_base_case( _  ) -> false.

solve_base_case( []) -> [];
solve_base_case([H]) -> [H].

divide(Lst) ->
    {L1, L2} = lists:split(length(Lst) div 2, Lst),
    SubProblems = [L1, L2],
    SubProblems.

combine(Solutions) -> [SL1, SL2] = Solutions,
                      lists:merge( SL1, SL2 ).
```

Ezt az *Eliminate Variable* transzformációnk továbbfejlesztésével még szebbé lehetne alakítani, de már ez is megfelelő ahhoz, hogy a párhuzamos *d&c* sémát bevezessük.

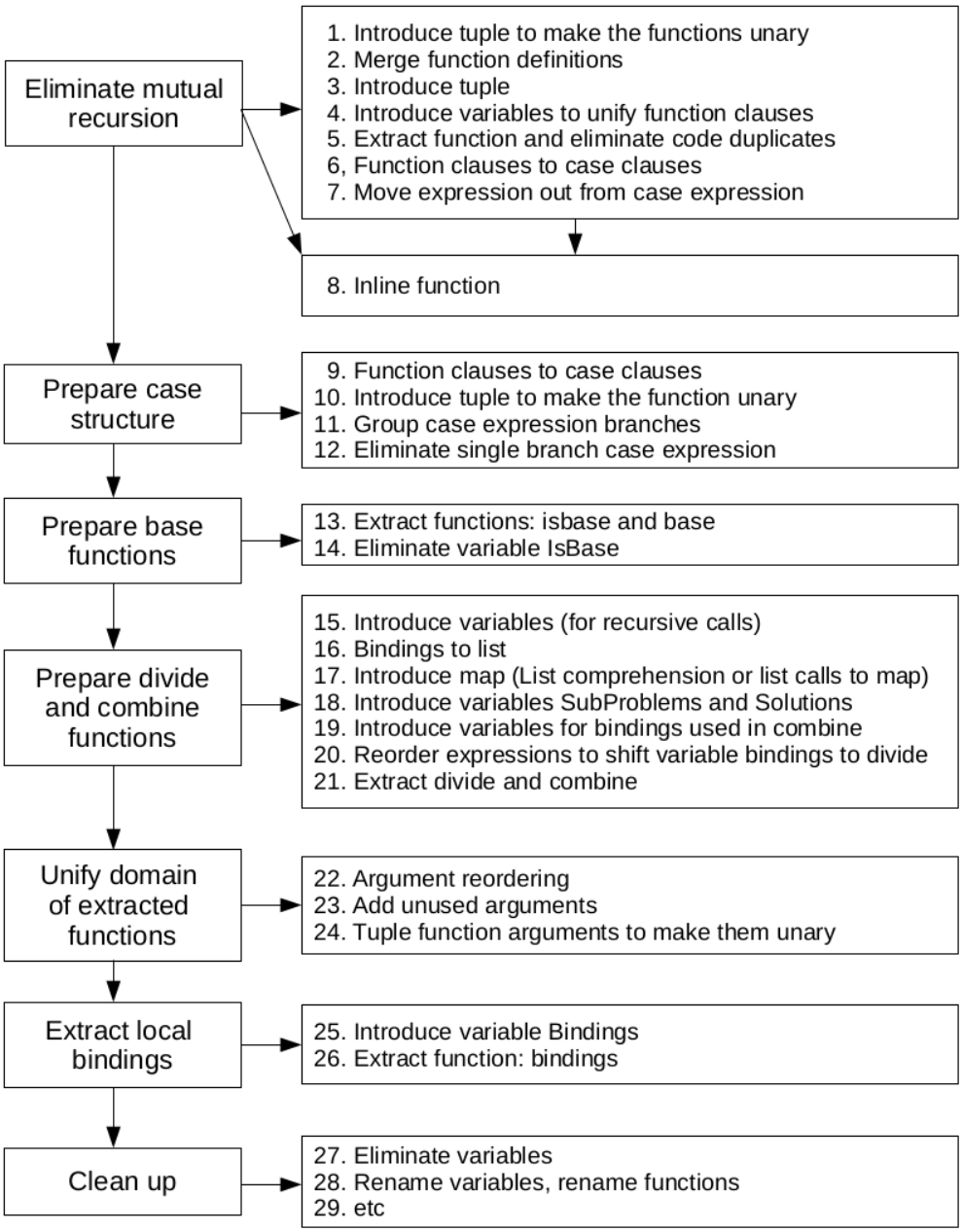
```
ms(Lst) ->
  (sk_hlp:dc_lim(  fun is_base_case/1,  fun solve_base_case/1,
                   fun divide/1,        fun combine/1,          16  )
  )(Lst).
```

A kanonikus alak általánosítása

Az ezidáig kanonikusnak tekintett alakot érdemes lehet picit általánosítani. Vannak olyan *d&c*-minták, amelyben a fenti kanonikus alak előállítása hatékonyságbeli romlást okoz a kódon. Erre az esetre javasoljuk azt az általánosítást, amely részeredmények megosztását teszi lehetővé a *d&c*-számítást alkotó függvények között.

```
dcgen(Problem) ->
  Bindings = bindings(Problem)
  case is_base_case(Bindings) of
    true  ->
      solve_base_case(Bindings);
    false ->
      {SubProblems,NewBindings} = divide(Bindings),
      Solutions = lists:map(fun dcgen/1, SubProblems),
      combine(Solutions,NewBindings)
  end.
```

Az általánosított kanonikus alak hasznát bemutathatjuk a Karatsuba-szorzásos példánkon (lásd a 18. oldalon). Pusztán az automatizált transzformációk ügyes megválasztásával és végrehajtásával eljuthatunk ehhez az alakhoz [27].



2.3. ábra. A *d&c*-refaktorálás módszertana

Összefoglalva az eddigieket, a *d&c*-mintának megfelelő függvényeket automatizált transzformációk emberi tervezés szerinti sorozatba rendezésével javasuljuk először kanonikus (vagy általánosított kanonikus) alakra hozni, majd a párhuzamos *d&c*-séma használatára átírni. A transzformációk megválasztásához támpontot a 2.3. ábra ad. A választás és a sorrend kialakítása kapcsán az emberi intelligencia mesterséges intelligenciára cserélése további kutatást igényel.

3. fejezet

Egyéb eredmények

Az előző két fejezetben bemutatott eredmények többsége a RefactorErl, illetve a belőle kifejlesztett PaRTE eszközzel kapcsolatos kutatásaimból származtak, így ezek az eredmények szorosan kapcsolódnak egymáshoz. Ehhez gyengébben, de kapcsolódik az ebben a fejezetben röviden összefoglalt számos egyéb kutatásom is, melyek programozási nyelvekre, konkurens, párhuzamos és elosztott programok készítésének eszközeire, programok elemzésére, feldolgozására irányultak.

3.1. Típusozás

A típuskikövetkeztetés problematikája a Core Erlang nyelv kapcsán már előkerült (l. 13. oldal). Doktoranduszommal, Góbi Attilával egy másik érdekes problémát vizsgáltunk: azt, hogy a típusozást hogyan lehetne függetlenebbé tenni a névfeloldástól [9, 8]. Az azonosítók egy programozási nyelvben több definícióra is utalhatnak egy adott kontextusban. Például egy rekordmező több rekordban is előfordulhat. Amikor hivatkozunk egy rekordmezőre, szükségünk lehet típusinformációkra ahhoz, hogy el tudjuk dönteni, melyik rekord mezőjére utal a használt azonosító (típusvezérelt névfeloldás, type-directed name resolution). Viszont a típusozás során is szükségünk van olyan információkra, hogy egy-egy adott név milyen típusú dolgot azonosít (típuskörnyezet, type environment). Emiatt a típusozás és a névfeloldás folyamata összekeveredik, mely nagyon bonyolult algoritmust eredményez. Ezt elkerülendő, a típuskikövetkeztetést intenzíven használó funkcionális nyelvekben megszorításokat tehetnek az azonosítók használatára (pl. Haskellben csak két különböző modulban szerepelhet ugyanaz az azonosító rekordmezőként), vagy korlátozzák a típuskikövetkeztetést (pl. Cleanben a rekord típusú kifejezések típusát deklarálni kell), vagy „elfogult” (biased) szabályokat hoznak (pl. OCamlben a deklarációk sorrendje számít).

Javasoltunk egy olyan módszert [9], amely a Bottom-Up algoritmus [HHS02] egy változatát használja típuskikövetkeztetésre. A Bottom-Up algoritmus kényszerfeltételeket (*constraints* és *assumptions*) gyűjt a típusozás első lépéseként. Ennek során még nincs szükség névfeloldásra. A mi módosításunk a típusvezérelt névfeloldást igénylő azonosítókra vonatkozó kényszerfeltételeket külön gyűjti, és csak akkor próbálja az ezen azonosítókra vonatkozó hivatkozásokat feloldani, amikor a többi kényszerfeltételt már megoldotta, és a megoldásból minden olyan típusinformációt, amely segíthet a névfeloldásban, már összegyűjtött. A módszer tehát lehetővé teszi, hogy a típusozás és a névfeloldás algoritmusa szétváljon, és a típuskényszerek megoldása, valamint a nevek feloldása egymást követő független lépések ismétlődő sorozataként valósuljanak meg.

Ezt az elgondolást továbbfejlesztve megadtunk egy típusellenőrzési és -kikövetkeztetési módszert az F# nyelv egy egyszerűsített változatához, az általunk G♭-nek keresztelt nyelvhez [8]. A típuskörnyezetet is szétbontottuk két részre (függvények és rekordmezők), és a Bottom-Up algoritmusba beletettük a deklarált típusú változók esetét. Az összegyűjtött kényszerfeltételeket olyan sorrendben oldjuk meg, hogy a rekordmezőkre vonatkozó hivatkozások feloldása előtt minél több típusinformációt ki tudjunk nyerni.

3.2. Erőforráselemzés és optimalizáció

Szintén Góbi Attila doktoranduszom doktori kutatásához kapcsolódóan, de Diviánszky Péterrel együttműködve vizsgáltunk egy módszert programok erőforrásfelhasználásának becslésére [7]. Egy magasabbrendű, mohó funkcionális, néhány beépített típussal bíró magnyelv esetére felépítettünk egy paraméterezhető modellt, amellyel pontos költséget lehet rendelni az egyes konstrukciókhoz egy tetszőleges nemcsökkenő erőforrás tekintetében. A modell alapján a magnyelv összetett konstrukciókat tartalmazó kiterjesztéseihez absztrakt interpretációval költségbecslő eljárást adtunk meg, mely költségintervallumokat határoz meg az egyes kifejezésekhez. A bemutatott módszer egyik hiányossága, hogy az optimalizációk hatását nem veszi figyelembe. Így például a valós idejű rendszerek programozásánál oly fontos WCET-becslésre (worst-case execution time, legrosszabb futási idő) csak korlátozottan alkalmazható.

Doktoranduszaimmal, Artyom Antyipinnel (akinek ez a kutatási témája) és Góbi Attilával kidolgoztunk egy olyan optimalizációs technikát, amely pontosabbá tudja tenni a WCET-elemzéseket, ráadásul a vizsgált architektúrákon jelentős hatékonyságbeli javulást eredményezett átlagos (nem rendezett) adatokon [3]. A technika lényege, hogy egy feltételtől függő ugró utasítás és egy ezzel védett adatmásolás (mely tipikus gépi kódja lehet egy if-then-else utasításnak) lecserélhető egy feltételes adatmásoló utasításra. Ezáltal a téves végrehajtásiút-előrejelzések (branch mis-prediction) kiküszöbölhetők. A téves előrejelzések bizonyos körülmények esetén jelentősen csökkentik a hatékonyságot, és – mivel költségük nehezen határozható meg – a WCET-elemzést is pontatlanabbá teszik. Egy adatsorozatok feltételes összegzését végző programmal végzett tesztjeinkben a *GCC* és a *Clang/LLVM* által előállított optimalizált kódhoz képest aszimptotikusan 2.6-szoros gyorsulást értünk el Intel Sandy Bridge és AMD K10/K12 architektúrán. Ezzel sikerült a gyakorlatban is alátámasztani az elméleti számításokat. Az optimalizáció eredményesen alkalmazható olyan többágú elágazó utasításoknál, amelyeknél a feltétel(ek) egy darab assembly összehasonlító utasítással van(nak) kódolva, az ágak összköltsége kisebb, mint a téves előrejelzés miatti veszteség, és a végrehajtott műveletekhez létezik neutrális érték.

Az ezzel az eredménnyel az OTDK-n 2. díjat nyert hallgatómmal, Juhász Dáviddal a szuperoptimalizáció [CC95, LGC02] lehetőségeit vizsgáltuk [17]. A szuperoptimalizáció lehetővé teszi azt, hogy egymástól független, de egymás eredményeit felhasználni képes optimalizációkat ne egymás után (vagy esetleg fixpontig iterálva), hanem összeolvasztva hajtsunk végre: ez az optimalizációk eredményességének javulásával járhat. A monolitikus szuperoptimalizáció viszont a fordítóprogram modularitását sértheti. Ennek elkerülése érdekében olyan keretrendszert lehet konstruálni, amelyben modulárisan elhelyezhetők az egyes optimalizációk, és a keretrendszer gondoskodik az összeolvasztásukról [LGC02]. Kutatásunkban egy ilyen keretrendszert hoztunk létre az LLVM fordítóprogram-infrastuktúrához. Általánosítottuk az *integrált elemzés* (integrated analysis [LGC02]) fogalmát, és kiterjesztettük az *összetett elemzés*eket (composed analysis [LGC02]) egy utasításról egy teljes programra. Az adatfolyamelemzésekhez hasonlóan implementáltuk az elkészült öt integrált elemzést, illetve ezekre alapozva az összetett elemzést az LLVM eszközeivel. Az integrált elemzések a hagyományos elemzésekhez képest másfélszer annyi kódsort igényeltek – ez az ára az általánosításnak –, de az elemzések moduláris jellege megmarad.

A statikus erőforráselemzés mellett dinamikus erőforráselemzéssel is foglalkoztam. Részt vettem egy projektben, amely ODF (Open Document Format) dokumentumok elosztott kezelését, mobil eszközökön történő kliens-szerver alapú szerkesztését támogató módszerek kifejlesztését tűzte ki célul [4, 11]. A mobileszközök korlátozott erőforrásaihoz illeszkedő dokumentumsémákat, réteges dokumentumreprezentációt, illetve ezeket támogató (prototípus) szoftvereszközöket dolgoztunk ki. Az én szerepem a különböző szoftvereszközök erőforrásigényét felmérő tesztelési forgatókönyv kidolgozása volt.

3.3. Alkalmazásiterület-specifikus nyelvek és speciális könyvtárak

Doktoranduszommal, Poór Artúrral, valamint további kollégákkal definiáltunk egy lekérdező nyelvet, mellyel a RefactorErl SPG-jéből kinyerhetünk statikus elemzési, szemantikai információkat egy Erlangban írt szoftverről [37]. Szemben a RefactorErl standard Szemantikus Lekérdező Nyelvével, melyben a lekérdezések úgy néznek ki, mint egy nyaklánc (szelektorok, szűrők és tulajdonságok pontokkal tagolt sorozatai), az itt javasolt NequeLace (New Query Language) inkább a halmazkészítő jelölésrendszerre épít – ezzel a szintaxissal számos más lekérdező nyelvben is találkozhatunk. A NequeLace nyelv szigorúan típusos, és a modularitás és az újrafelhasználhatóság érdekében lekérdező függvények definiálását is lehetővé teszi.

2010-től 2012-ig az *Elosztott és sokmagos rendszerek szoftvertechnológiai kérdései* „alprojektnek”[1] voltam a vezetője, azon belül pedig *Sokmagos rendszerek programozása* kutatócsoport egyik vezető kutatója. Csoportunkat érdekelték a nagy, akár potenciálisan végtelen hosszú, vagy aszinkron módon előálló adatsorozatokon értelmezett számítások, melyek párhuzamos végrehajtása akkortájt a Big Data igen népszerű módszerévé vált. Egy cikkünkben például a C++ Standard Template Library (STL) eszközeit bővítettük ki a *végtelen felsoroló* (infinite iterator) egy általános, generátorfüggvényen alapuló megvalósításával [24]. Doktoranduszommal, Góbi Attilával még tovább mentünk a stream-orientált programozás vizsgálatában [10], és ehhez az inspirációt Ralph Hinze egy cikke szolgáltatta [Hin10], valamint a C++11 új lehetőségei. Ebben a megközelítésben a Fibonacci-számokat reprezentáló végtelen sorozatot így hozhatjuk létre.

```
stream<int> fib = 0 <<= fib + (1 <<= fib);
```

Az `std::stream` osztállyal szemben a mi `stream` sablonosztályunk egy önhivatkozást lehetővé tevő típus, mely egy számítási gráffal ábrázolja az adatstruktúrát – ezt a gráfot a műveletek gráfátírással módosítják a kiértékelés folyamán. Megoldásunkkal elegánsan, deklaratív stílusban írhatunk le számításokat – nem csak lusta funkcionális, vagy korutinokat támogató nyelvekben, de akár C++-ban is. Egy dinamikus programozási feladat esetében az implementációnk hatékonysága a *memoization*re *unordered map* adatszerkezetet használó szokványos C++ implementációnál 50%-kal lassabb, de a *tree map* adatszerkezetet használónál 50%-kal gyorsabbnak bizonyult. A streamek megvalósítása során építettünk a C++ számos új lehetőségére, mint pl. koinduktív definíciók részlegesen inicializált adatokkal történő ábrázolása, *lvalue* és *rvalue* hivatkozások feletti polimorfizmus, illetve felhasználó által bevezetett literálok. Később doktoranduszaimmal, Artyom Antyipinnel és Góbi Attilával egy másik módszert is kidolgoztunk lusta definíciójú objektumok manipulálására [2]. A lustaságot itt függvényobjektumok és lambdakifejezések segítségével valósítjuk meg, melyeket speciális *megosztott mutatók* (shared pointer) mögé rejtünk a biztonságos használat érdekében. Ezzel a módszerrel is leírhatunk koadatokat, pl. streameket, de nem olyan elegáns a használatuk, mint az előző módszerrel.

Ugyanebben a kutatásban a C++ STL párhuzamosításának módszereit is vizsgáltuk, itt is felhasználva a C++11 új lehetőségeit [39]. Három megoldást dolgoztunk ki, melyekkel (1) a pipeline párhuzamos séma, (2) a spekulatív számítás sémája, valamint (3) az asszociatív műveletek párhuzamos oszd-meg-és-uralkodj elvű kiszámításának sémája vált elérhetővé az STL használója számára. A pipeline sémához a `for_each` függvénynek átadandó függvényobjektumot tudjuk egy párhuzamosan kiértékelt függvényobjektum-kompozícióval kifejezni. A spekulatív számítások támogatásához a logikai *és/vagy* műveletek operandusainak párhuzamos kiértékelésére vezettünk be függvényobjektumokat, melyekkel a `find_if`, `count_if` stb. algoritmusokban a feltételek kiértékelését lehet felgyorsítani. Az asszociatív műveletek kiszámításának sémájához az `accumulate` algoritmust módosítottuk úgy, hogy érzékelni tudja egy extra `associative` paraméter jelenlétét a kapott bináris műveletben, és ennek megfelelően egy adott thresholdig párhuzamosan számolja a műveletet a kapott adatsorozaton.

[1] Az *Európai Léptékkel a Tudásért, ELTE* című projektben (TÁMOP 4.2.1./B-09/1/KMR-2010-000).

A doktori értekezésem benyújtása előtt elkezdtünk egy kutatást, mely számítási feladatok (*job*ok) grid infrastruktúrán történő optimális ütemezésének módszertanát vizsgálta (lásd pl. [LKUH05], bár ezek az eredmények nem jelentek meg a doktori értekezésemben). Javasoltunk egy módszert, mely jobok tulajdonságait leírni képes alkalmazásiterület-specifikus nyelvet (domain specific language, DSL) állított fókuszba. Ezt a kutatást később is folytattunk [30, 47, 48]. Ennek során finomítottuk a jobleíró nyelvet, és bevezettük az összetett jobleírásokat, melyek több lefutás során összegyűjtött statisztikák alapján javítja az egyszerű jobleírásokat, figyelembe véve esetleg a job vezérlésifolyam-gráfjában a különböző végrehajtási utakhoz tartozó erőforrásfelhasználási tulajdonságokat.

A *CPS Programozás* munkacsoport[2] egyik vezetőjeként 2012-től 2013-ig részt vettem egy kiber-fizikai rendszerek (cyber-physical system, CPS) programozását megkönnyítő módszer kidolgozásában [23]. Célunk CPS-ek ún. *munkafolyamat*tal (workflow) történő leírásának támogatása volt egy alkalmazásiterület-specifikus programozási nyelven keresztül. A munkafolyamatok olyan (gyengén mellékhatásmentes [44]) modulokból épülnek fel, (1) melyek között a kölcsönhatásokat minimalizáljuk, és (2) amely modulokat hierarchiába szervezzük, valamint (3) amelyeket monitorozni is tudunk, hogy a sikertelenséget, illetve a tervezett működéstől való eltérést detektálni, modell-alapú előrejelzés alapján hibajavító modulok indításával kompenzálni lehessen. A munkafolyamatok tesztelhetősége, verifikálhatósága, valamint kényszerfeltételeknek (pl. WCET) való megfelelése alapvető fontosságú – a hierarchikusság és a (legalább gyenge) mellékhatásmentesség ezt a célt szolgálja. A DSL-hez a Task-Oriented Programming paradigma [PLM+12] szolgáltatta az alapötletet. A szenzorok olvasása, az aktuátorok működtetése, valamint az elemi számítási feladatok ún. primitív taszkokként jelennek meg, melyeket különféle kombinátorokkal szervezhetünk hierarchiába, figyelhetünk meg, illetve ruházhatunk fel kényszerfeltételekkel. A DSL-t először az Erlang nyelv egy kiterjesztésébe ágyaztuk be, ahol a kiterjesztés (számítási csomópontok között hálózaton átküldhető függvénydefiníciók, speciális operátorok és kulcsszavak) megvalósításához a RefactorErl eszközt használtuk [JDK15], illetve később egy Scala alapú megvalósítás is készült.

3.4. Programok szemantikája és helyessége

Még a doktori kutatásom során Horváth Zoltánnal, Peter Achtennel és Rinus Plasmeijerrel elkezdtünk foglalkozni interaktív, valamint kódmobilitást tartalmazó funkcionális programok olyan tulajdonságainak megfogalmazásával és bizonyításával, amely tulajdonságok az általunk művelt, a konkurens programozásban megszokott fogalmakra vezethetők vissza [HAKP99a, HAKP99b]. Az én doktori értekezésem más irányba kanyarodott, de ez a kutatás alapozta meg Tejfel Máté doktori kutatását, melyben én is részt vettem, és amely során bevezettük például az objektumabsztrakció fogalmát [THK05]. Az ez alapján vett helyességbizonyításhoz a Sparkle [dMvEP02] rendszert használtuk, melyhez egy kiegészítés, a Sparkle-T is elkészült [THK06].

Fokozatom megszerzése után tovább folytatódott ez a kutatás, melyben egyes technikai részletek kidolgozásával, valamint tanácsadással vettem részt. Az egyik eredményünk egy olyan módszer kidolgozása volt, amellyel interaktív, grafikus felületű funkcionális programokról tudunk érvelni [43]. Az ilyen programok olyan nyelvi elemeket használnak, és olyan komplex programkönyvtárakra támaszkodnak, hogy a Sparkle bizonyítórendszer képtelen ezeket kezelni. Ezért létrehoztunk egy olyan modellt, amelyben szimulálni lehet a szóban forgó alkalmazásokat, és amely modell axiómák formájában betölthető a bizonyítórendszerbe. A módszerünket egy több folyamatból álló, eseményvezérelt alkalmazás tulajdonságainak vizsgálatával illusztráltuk. Megmutattuk, hogy az Object I/O könyvtárat használó Clean programok kis módosításokkal (pl. típusok törlése vagy egyszerűsítése) átírhatók a modellünk, a Simplified IO (Sio) használatára,

[2] A hazai finanszírozású EITKIC 12-1-2012-0001 projektben.

és így elvégezhetők a szükséges helyességbizonyítások a Sparkle-ben.

A korábban csak axiomatikus szemantikával vizsgált objektumabsztrakcióhoz kidolgoztunk egy operációs szemantikát, melynek az *annotated tree-semantics* nevet adtuk. Ez a szemantika lehetőséget ad arra, hogy a korábban használt axiomatikus szemantika helyességét megvizsgáljuk, és így tegyük pontosabbá a funkcionális programokban előforduló, logikailag összetartozó értékek tulajdonságainak egyfajta temporális logika szerinti elemzését [41, 40]. Kiinduláshoz a Clean nyelv gráfátíráson alapuló operációs szemantikáját használtuk. Egy Clean programhoz egy gráfátírási lépéssorozat adható meg, amelyet a Clean (többnyire lusta, helyenként mohó) kiértékelési szabályai határoznak meg. Ha eltekintünk ettől a sorrendtől, a programokhoz lépéssorozat helyett egy levezetési fát rendelhetünk, melynek csomópontjai programgráfok, irányított élei pedig a kiértékelés lehetséges irányait mutatják. Az annotated tree-semantics alapján megjelöljük azokat az értékeket, amelyek a program kiértékelése során egy absztrakt objektum egyes állapotait reprezentálják: ezek a jelölések rákerülnek a levezetési fa megfelelő csomópontjaiban szereplő gráfok azon csomópontjára, amelyek valamelyik szóban forgó értéket írják le. Az objektumabsztrakciós jelöléseket tartalmazó levezetési fa egyes élei az objektumok létrehozását, más élei pedig az objektumok (gyakran együttes, szinkronizált) állapotátmeneteit jelentik. A részleteket mellőzve azt mondhatjuk, hogy az így kapott annotált levezetési fa, valamint az objektumlétrehozást és állapotátmeneteket reprezentáló élek segítségével megfogalmazhatók és vizsgálhatók a program absztrakt objektumaira elágazó idejű temporális logikai állítások. Ennek a módszernek a formális, precíz kidolgozását Tejfel Máté doktori értekezése tartalmazza [Tej08].

Tovább folytattuk a mobil (azaz programfutás közben továbbított és beépülő) kódokat használó programok helyességvizsgálatát támogató módszerek fejlesztését is. Kidolgoztunk egy esettanulmányt a Clean nyelv Dynamic konstrukciójával létrehozott funkcionális programok viselkedéséről érvelő módszerhez [42], amely egy újabb lépést jelentett a „Bizonyított tulajdonság tanúsítványát hordozó kód” (Certified Proved Property Carrying Code, CPPCC) technika [HK02, DHK02] – mely a Proof-Carrying Code [Nec97] egy továbbfejlesztett változata – kiteljesítése felé. Hasonló módszereket vizsgáltunk Istenes Zoltánnal és hallgatóinkkal a Java virtuális gép kapcsán [16]. Felépítettünk egy CPPCC-keretrendszert, mely a Java/JML és Spark/Ada nyelvekre, valamint a B-módszerre [Abr96] és az ESC/Java2 eszközre [BCC+05] épített. Erre a CPPCC-keretrendszerre építve kidolgoztunk egy módszert robotok mobil kóddal történő távoli irányításához, és bemutattuk a módszert egy esettanulmányon [15, 14]. Ebben az esettanulmányban egy NXT-robotot irányítottunk B-módszerrel kifejlesztett, (számítógéppel ellenőrizhetően) bizonyítottan helyes szoftverrel.

Hivatkozott publikációim

[1] M. Alqaradaghi and T. Kozsik. Switch to the new switch in Java. In *Abstract book for the 16th Miklós Iványi Int'l PhD & DLA Symp.*, page 207. University of Pécs, 2020. Paper 143.

[2] A. Antyipin, A. Góbi, and T. Kozsik. Manipulating infinite data in c++ using lazy shared objects. In *Proc. CSE 2012 Int'l Scientific Conf. on Computer Scinece and Engineering*, pages 103–110. Technical University of Kosice Faculty of Electrical Engineering and Informatics, 2012.

[3] A. Antyipin, A. Góbi, and T. Kozsik. Low level conditional move optimization. *ACTA CYBERNETICA-SZEGED*, 21:5–20, 2013.

[4] I. Barna, P. Bauer, K. Bernád, Z. Hernáth, Z. Horváth, B. Kőszegi, G. Kovács, T. Kozsik, Z. Lengyel, R. Roth, S. Sike, and G. Takács. ODF Mobile Edition – Towards the development of a mobile office software. In *Proc. 8th Int'l Conf. on Applied Informatics, Eger, Hungary, January 27–30, 2010*, volume 2, pages 313–321, 2010.

[5] I. Bozó, V. Fördős, D. Horpácsi, Z. Horváth, T. Kozsik, J. Kőszegi, and M. Tóth. Refactorings to Enable Parallelization. *LECTURE NOTES IN COMPUTER SCIENCE*, 8843:104–121, 2015.

[6] I. Bozó, V. Fördős, Z. Horváth, M. Tóth, D. Horpácsi, T. Kozsik, J. Kőszegi, A. Barwell, C. Brown, and K. Hammond. Discovering parallel pattern candidates in Erlang. In L. M. Castro and H. Svensson, editors, *Proc. 13th ACM SIGPLAN workshop on Erlang*, pages 13–23, New York, 2014. ACM Press.

[7] P. Diviánszky, A. Góbi, and T. Kozsik. Size analysis of higher-order functions. pages 23–37, Madrid, 2011. Tech. Rep. SIC-08/11, Dept. Computer Systems and Computing Universidad Complutense de Madrid.

[8] A. Góbi and T. Kozsik. Type inference in G♭. In H. F. Pop and A. Bege, editors, *Selected Papers of 8th Joint Conf. on Mathematics and Computer Science, July 14–17, 2010, Komárno, Slovakia*, pages 197–210, Győr, 2011. Novadat.

[9] A. Góbi, T. Kozsik, M. Mészáros, A. Antyipin, D. Batha, and T. Kiss. Untangling type inference and scope analysis. In *Proc. 8th Int'l Conf. on Applied Informatics*, volume 2, pages 157–164, Eger, 2010. Eszterházy Károly College.

[10] A. Góbi, Z. Szűgyi, and T. Kozsik. A c++ pearl: self referring streams. *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE SECTIO COMPUTATORICA*, 37:157–174, 2012.

[11] Z. Horváth, I. Barna, P. Bauer, K. Bernád, Z. Hernáth, B. Kőszegi, G. Kovács, T. Kozsik, Z. Lengyel, R. Roth, S. Sike, and G. Takács. A client-server model for editing

ODF documents on mobile devices. *ACTA ELECTROTECHNICA ET INFORMATICA*, 11(3):17–20, 2011.

[12] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. Nagyné Víg, T. Nagy, M. Tóth, and R. Király. Building a refactoring tool for Erlang. In K. Mens, M. v. d. Brand, A. Kuhn, H. M. Kienle, and R. Wuyts, editors, *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, 2008. 11 pages.

[13] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. Víg, T. Nagy, M. Tóth, and R. Király. Modeling semantic knowledge in Erlang for refactoring. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 54:7–16, 2009. Special issue for Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Cluj-Napoca (Romania), Volume I.

[14] Z. Istenes and T. Kozsik. Safe mobile code controlling a robot. In C. Attiogbé and D. Kröning, editors, *Proc. 1st Int'l Workshop on Property Verification for Software Components and Services, ProVeCS (Satellite of TOOLS Europe)*, pages 26–37, 2007. ETH Technical Report 567.

[15] Z. Istenes and T. Kozsik. Commanding a robot in a safe way. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae Sectio Computatorica*, 30:175–190, 2009.

[16] Z. Istenes, T. Kozsik, C. Hoch, and A. L. Tóth. Proving the correctness of mobile Java code. *Pure Mathematics and Applications*, 17(3-4):323–342, 2006.

[17] D. Juhász and T. Kozsik. Superoptimization in LLVM. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae Sectio Computatorica*, 36:179–199, 2012.

[18] R. Kitlei, I. Bozó, T. Kozsik, M. Tejfel, and M. Tóth. Analysis of preprocessor constructs in Erlang. In S. L. Fritchie and K. F. Sagonas, editors, *Erlang '10: Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, pages 45–55, New York, 2010. ACM Press.

[19] R. Kitlei, L. Lövei, T. Nagy, Z. Horváth, and T. Kozsik. Layout preserving parser for refactoring in Erlang. *ACTA ELECTROTECHNICA ET INFORMATICA*, 9(3):54–63, 2009.

[20] T. Kozsik. Let's PaRTE! Talk at Software Technology Forum, 2015.

[21] T. Kozsik. Refactoring for parallelization. Invited talk at 12th Joint Conf. on Mathematics and Computer Science, Cluj, Romania, 2018.

[22] T. Kozsik, Z. Csörnyei, Z. Horváth, R. Király, R. Kitlei, L. Lövei, T. Nagy, M. Tóth, and A. Víg. Use cases for refactoring in Erlang. *Lecture Notes in Computer Science*, 5161:250–285, 2008.

[23] T. Kozsik, A. Lőrincz, D. Juhász, L. Domoszlai, D. Horpácsi, M. Tóth, and Z. Horváth. Workflow description in cyber-physical systems. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 58(2):20–30, 2013.

[24] T. Kozsik, N. Pataki, and Z. Szűgyi. C++ standard template library by infinite iterators. *ANNALES MATHEMATICAE ET INFORMATICAE*, 38(1):75–86, 2011. ScopusID: 84856351458.

[25] T. Kozsik and M. Tóth. Let's PaRTE! *COMPUTERWORLD SZÁMÍTÁSTECHNIKA*, XLVI(2):16–17, 2015.

[26] T. Kozsik, M. Tóth, and I. Bozó. Divide a Divide-and-Conquer into a Divide and a Conquer. Invited talk at 10th Int'l Conf. on Applied Informatics, Eger, 2017.

[27] T. Kozsik, M. Tóth, and I. Bozó. Free the Conqueror! Refactoring divide-and-conquer functions. *Future Generation Computer Systems*, 79(P2):687–699, 2017.

[28] T. Kozsik, M. Tóth, I. Bozó, and Z. Horváth. Static analysis for divide-and-conquer pattern discovery. *COMPUTING AND INFORMATICS*, 35(4):764–791, 2016.

[29] H. Li, S. J. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang programs. In *Proc. 12th Int'l Erlang/OTP User Conference, Stockholm*, 2006. 10 pages.

[30] L. Lőrincz, A. Ulbert, Z. Horváth, and T. Kozsik. Towards an agent integrated speculative scheduling service. In *Distributed and Parallel Systems: From Cluster to Grid Computing*, pages 211–222. Springer, New York, 2007.

[31] L. Lövei, Z. Horváth, T. Kozsik, and R. Király. Introducing records by refactoring. In *Proc. 2007 SIGPLAN workshop on ERLANG*, pages 18–28. ACM, 2007.

[32] L. Lövei, Z. Horváth, T. Kozsik, R. Király, and R. Kitlei. Static rules of variable scoping in Erlang. In *Proc. 7th Int'l Conf. on Applied Informatics*, pages 137–145, Eger, 2007. Eszterházy Károly Tanárképző Főiskola (EKTF).

[33] L. Lövei, Z. Horváth, T. Kozsik, R. Király, A. Víg, and T. Nagy. Refactoring in Erlang, a dynamic functional language. In D. Dig, editor, *Proc. 1st Workshop on Refactoring Tools (WRT'07)*, pages 45–46. Technische Universität Berlin, 2007.

[34] L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang programs. *PERIODICA POLYTECHNICA-ELECTRICAL ENGINEERING*, 51(3-4):75–84, 2007. ScopusID: 76049110560.

[35] G. Oláh, D. Horpácsi, T. Kozsik, and M. Tóth. Type inference for Core Erlang to support test data generation. *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, 59(Special Issue 1):201–215, 2014.

[36] ParaPhrase WP2. *Side Condition Analysis*, 2014. (Deliverable D2.12, Work Package 2, Task 2.4), D. Horpácsi and T. Kozsik (eds.), 25 pages.

[37] A. Poór, I. Bozó, T. Kozsik, G. Páli, and M. Tóth. Benefits of implementing a query language in purely functional style. In *Proc. 11th Joint Conf. on Mathematics and Computer Science*, volume 2046 of *CEUR Workshop Proceedings*, pages 250–266, 2016.

[38] A. Poór, T. Kozsik, M. Tóth, and I. Bozó. Compiler front end fusion – undo desugaring in language processing tools. *STUDIA UNIV. BABEȘ–BOLYAI, INFORMATICA*, LXIII(2):5–20, 2018.

[39] Z. Szűgyi, M. Török, N. Pataki, and T. Kozsik. High-level multicore programming with C++11. *Computer Science and Information Systems*, 9(3):1187–1202, 2012.

[40] M. Tejfel and T. Kozsik. An operational semantics of temporal properties in functional programs. In *Draft Proc. 9th Symposium on Trends in Functional Programming (TFP)*, pages 396–406, 2008. Technical report ICIS-R08007, Radboud University Nijmegen.

[41] M. Tejfel, T. Kozsik, and Z. Horváth. An interpretation of temporal properties in functional programs. In C. Olaf, editor, *Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007*, pages 224–228, 2007. Extended Abstract.

[42] M. Tejfel, T. Kozsik, and Z. Horváth. Object based multiparadigm concepts for verification of functional components. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008) at European Conference on Object-Oriented Programming (ECOOP 2008)*, 2008.07.07. Art. No.: 2.6.

[43] M. Tejfel, T. Kozsik, and Z. Horváth. A semantic model for proving properties of Clean Object I/O programs. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae Sectio Computatorica*, 31:107–147, 2009.

[44] Z. Tősér, R. Bellon, D. Hornyák, Z. Hotváth, T. Kozsik, R. Rill, and A. Lőrincz. Functional programming framework for cyber-physical systems. In *Proc. Knowledge Engineering: Principles and Techniques*, pages 21–24, Cluj-Napoca, Romania, 2015.

[45] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik. Impact analysis of Erlang programs using behaviour dependency graphs. *LECTURE NOTES IN COMPUTER SCIENCE*, 6299:372–390, 2010.

[46] M. Tóth, I. Bozó, and T. Kozsik. Pattern candidate discovery and parallelization techniques. In N. Wu, editor, *Proc. 29th Symposium on the Implementation and Application of Functional Programming Languages*, pages 1–26, New York, 2018. ACM.

[47] A. Ulbert, L. C. Lőrincz, T. Kozsik, and Z. Horváth. Speculative scheduling of parameter sweep applications using job behavior descriptions. *International Journal of Grid and High Performance Computing*, 1(1):22–38, 2009.

[48] A. Ulbert, L. C. Lőrincz, T. Kozsik, and Z. Horváth. Speculative scheduling of parameter sweep applications using job behaviour descriptions. In E. Udoh, editor, *Cloud, Grid and High Performance Computing: Emerging Applications*, pages 72–89. Information Science Reference, Hershey, 2011. Chapter 5.

Irodalomjegyzék

[Abr96] J-R. Abrial, editor. *The B-Book*. Cambridge University Press, 1996.

[ACD98] Marco Aldinucci, Massimo Coppola, and Marco Danelutto. Rewriting Skeleton Programs: How to Evaluate the Data-Parallel Stream-Parallel Tradeoff. In S. Gorlatch, editor, *Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 44–58, Germany, May 1998.

[ACD+16] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool Evolution: A parallel pattern for evolutionary and symbolic computing. *International Journal of Parallel Programming*, 44(3):531–551, 2016.

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *Proc. 2006 ACM SIGPLAN Workshop on Erlang, ERLANG'06*, pages 2–10, New York, USA, 2006. ACM.

[AS13] Stavros Aronis and Konstantinos Sagonas. On using Erlang for parallelization. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, pages 295–310, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93*, pages 31–41, New York, USA, 1993. ACM.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'94*, pages 163–173, New York, USA, 1994. ACM.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BD77] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

[BDH+14] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. Cost-directed refactoring for parallel Erlang programs. *International Journal of Parallel Programming*, 42(4):564–582, Aug 2014.

[BLH12] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In Ricardo Peña and Rex Page, editors, *Trends in Functional Programming*, pages 82–97. Springer Berlin Heidelberg, 2012.

[Boz18] István Bozó. *Erlang programok statikus elemzése és szeletelése.* PhD thesis, Eötvös Loránd Tudományegyetem, Informatika Doktori Iskola, 2018.

[BV99] J. Barklund and R. Virding. Erlang 4.7.3 Reference Manual. `http://www.erlang.org/download/erl_spec47.ps.gz`, 1999.

[CC95] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[Col04] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.*, 30(3):389–406, March 2004.

[DHK02] Károly Daxkobler, Zoltán Horváth, and Tamás Kozsik. A prototype of CPPCC – safe functional mobile code in Clean. In *Draft proc. 14th Int'l Workshop on the Implementation of Functional Languages*, Madrid, Spain, szeptember 2002.

[Dig11] Danny Dig. A Refactoring Approach to Parallelism. *IEEE Softw.*, 28:17–22, 2011.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proc. 31st Int'l Conf. on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[dMvEP02] Maarten de Mol, Marco van Eekelen, and Rinus Plasmeijer. Theorem Proving for Functional Programmers, Sparkle: A Functional Theorem Prover. *Lecture Notes in Computer Science*, 2312:55–72, 2002. Springer-Verlag.

[DSNH04] P. Diviánszky, R. Szabó-Nacsa, and Z. Horváth. Refactoring via Database Representation. In *6th Int'l Conf. on Applied Informatics (ICAI 2004), Eger*, volume 1, pages 129–135, 2004.

[FK95] B. Freisleben and T. Kielmann. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computing and Informatics*, 14:579–596, 1995.

[Gar05] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

[GVL10] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, 2010.

[HAB+13] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Sscience*, pages 218–236. Springer, Berlin, Heidelberg, 2013.

[HAKP99a] Zoltán Horváth, Peter Achten, Tamás Kozsik, and Rinus Plasmeijer. Proving the temporal properties of the unique world. In Jaan Penjam, editor, *Software Technology, Fenno-Ugric Symposium (FUSST'99)*, pages 113–125, Tallinn, Estonia, augusztus 1999. Technical Report CS 104/99.

[HAKP99b] Zoltán Horváth, Peter Achten, Tamás Kozsik, and Rinus Plasmeijer. Verification of the temporal properties of dynamic Clean processes. In C. Clack and P. Koopman, editors, *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99)*, pages 203–218, Lochem, The Netherlands, szeptember 1999. draft.

[Her01] C.A. Herrmann. *The Skeleton Based Parallelization of Divide and Conquer Recursions*. Logos-Verlag, 2001.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Utrecht University, 2002.

[Hin10] Ralph Hinze. Reasoning about codata. In *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 42–93, 2010.

[HK02] Zoltán Horváth and Tamás Kozsik. Safe mobile code - cppcc: Certified proved-property-carrying code. In Juan Hernández and Ana Moreira, editors, *Object-Oriented Technology. ECOOP 2002 Workshop Reader*, volume 2548 of *Lecture Notes in Computer Science*, pages 8–10. Springer-Verlag Wien, 2002. As a part of: Grzegorz Czajkowski and Jan Vitek, Resource Management for Safe Languages (pp. 1-14).

[HKT16] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. Towards trustworthy refactoring in Erlang. In Geoff Hamilton, Alexei Lisitsa, and Andrei P. Nemytykh, editors, *Fourth International Workshop on Verification and Program Transformation*, volume 216, pages 83–103, Eindhoven, The Netherlands, July 2016.

[Hor18] Dániel Horpácsi. *Programtranszformációk helyessége és alkalmazásai*. PhD thesis, Eötvös Loránd Tudományegyetem, Informatika Doktori Iskola, 2018.

[JBH15] Vladimir Janjic, Christopher Brown, and Kevin Hammond. Lapedo: Hybrid skeletons for programming heterogeneous multicore machines in Erlang. In *Parallel Computing: on the Road to Exascale*, volume 27 of *Advances in Parallel Computing*, pages 185–195, 2015.

[JDK15] Dávid Juhász, László Domoszlai, and Barnabás Králik. Rea: Workflows for cyber-physical systems. In *Central European Functional Programming School*, volume 8606 of *Lecture Notes in Computer Science*, pages 479–506. Springer, 2015.

[LGC02] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *Proc. 29th ACM SIGPLAN-SIGACT Symp. on Principles Of Programming Languages*, pages 270–282, New York, USA, 2002. ACM.

[LKUH05] László Csaba Lőrincz, Tamás Kozsik, Attila Ulbert, and Zoltán Horváth. A method for job scheduling in Grid based on job execution status. *Multiagent and Grid Systems (MAGS), IOS Press*, 1(3):197–208, 2005.

[Loo12] Rita Loogen. Eden – Parallel Functional Programming with Haskell. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 142–206. Springer, Berlin Heidelberg, 2012.

[LS06] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP'06*, pages 167–178, New York, USA, 2006. ACM.

[LT08] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *WRT '08: Proc. 2nd Workshop on Refactoring Tools*, pages 1–4, New York, USA, 2008. ACM.

[MFM09] Shane A. Markstrum, Robert M. Fuhrer, and Todd D. Millstein. Towards Concurrency Refactoring for x10. In *Proc. 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '09, pages 303–304, New York, USA, 2009. ACM.

[MH88] Z.G. Mou and P. Hudak. An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2(3), 1988.

[MIK97] Greg Michaelson, Andrew Ireland, and Peter King. Towards a Skeleton Based Parallelising Compiler for SML. In *Proc. 9th Int'l Workshop on Implementation of Functional Languages*, pages 539–546, St. Andrews, Scotland, UK, 1997.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *SIGPLAN Not.*, 32(8):136–149, 1997.

[Nec97] G. Necula. Proof-carrying code. In *Proc. ACM Symposium on Principles of Programming Languages*, volume 2789 of *LNCS*, pages 106–119. Springer-Verlag, 1997.

[Nys03] Sven-Olof Nyström. A soft-typing system for Erlang. In *Proc. 2003 ACM SIGPLAN Workshop on Erlang, ERLANG'03*, pages 56–71, New York, USA, 2003. ACM.

[PDS10] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, pages 639–650. Springer Berlin Heidelberg, 2010.

[PLM+12] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-oriented programming in a pure functional language. In *Proc. 14th Symposium on Principles and Practice of Declarative Programming, PPDP'12*, pages 195–206. ACM, 2012.

[PS11] M. Pitidis and K. Sagonas. Purity in erlang. Proc. 22nd Int'l Conf. on Implementation and Application of Functional Languages*, Lecture Notes in Computer Science,Springer Berlin, Heidelberg*, 6647:137–152, 2011.

[TDDN00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int'l Symp. on Principles of Software Evolution (ISPSE'00)*, pages 157–167. IEEE Computer Society Press, 2000.

[Tej08] Máté Tejfel. *Funkcionális programozási nyelvek helyességvizsgálata.* PhD thesis, Eötvös Loránd Tudományegyetem, Informatika Doktori Iskola, 2008.

[THK05] Máté Tejfel, Zoltán Horváth, and Tamás Kozsik. Extending the Sparkle Core language with object abstraction. *Acta Cybernetica*, 17:419–445, 2005.

[THK06] M Tejfel, Z Horváth, and T Kozsik. Temporal properties of Clean programs proven in Sparkle-T. *LECTURE NOTES IN COMPUTER SCIENCE*, 4164:168–190, 2006.

[Tót18] Melinda Tóth. *Adat és kiértékelési függőségi elemzés funkcionális nyelvekre – Erlang programok statikus elemzése.* PhD thesis, Eötvös Loránd Tudományegyetem, Informatika Doktori Iskola, 2018.

[TuB+10] G. Tóth, P. Hegedűs, Á. Beszédes, T. Gyimóthy, and J. Jász. Comparison of different impact analysis methods and programmer's opinion: an empirical study. In *Proc. 8th Int'l Conf. on Principles and Practice of Programming in Java*, pages 109–118, New York, USA, 2010. ACM.

[VBF04] L. Vidacs, A. Beszedes, and R. Ferenc. Columbus schema for C/C++ preprocessing. In *Proc. 8th European Conf. on Software Maintenance and Reengineering*, pages 75–84, 2004.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6):156–165, 1993.

[ZHH06] V. Zsók, Z. Hernyák, and Z. Horváth. Designing distributed computational skeletons in D-Clean and D-Box. In *Central European Functional Programming School*, volume 4146 of *Lecture Notes in Computer Science*, pages 223–256. Springer, 2006.