

TDK-dolgozat

Oláh Gábor



EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
PROGRAMOZÁSI NYELVEK ÉS
FORDÍTÓPROGRAMOK TANSZÉK

Erlang programok típusozása¹

Témavezető:

Tóth Melinda

Tudományos segédmunkatárs
ELTE-IK, PNYF

Szerző:

Oláh Gábor

Programtervező matematikus
nappali tagozat, V. évfolyam

Budapest, 2011.

¹Támogatta a KMOP-1.1.2-08/1-2008-0002, Európai Regionális Fejlesztési Alap (ERFA) és az Ericsson Magyarország.

Tartalomjegyzék

1. Bevezetés	1
2. Hindley-Milner algoritmusa	3
2.1. Típusrendszer	4
2.2. Típuszabályok	6
2.3. A W algoritmus	7
3. Erlang, mint funkcionális nyelv	9
3.1. A Dialyzer	10
4. Success typing	12
4.1. Altípusrendszerek	13
4.2. Az Erlang magnyelv	15
4.3. Success typing fogalma	17
4.4. Korlátozás generálás és operációs szemantika	17
4.5. Az egyesítő algoritmus	21
5. A success typing Erlanghoz	24
5.1. Korlátozásgenerálás	25
5.2. Egységesítés	33
5.3. Példa típusozás	33
5.3.1. Korlátozásgenerálás	34
5.3.2. Második példa	34
6. Típuskikövetkeztető rendszer a RefactorErlben	37
6.1. Első prototípus	37

6.2. A success typing-on alapuló prototípus	38
6.3. Eredmények a RefactorErlben	41
6.3.1. Összehasonlítás a TypEr-rel	42
6.4. Fejlesztési lehetőségek	43
7. Összefoglalás	45

Kivonat

Az Erlang egy dinamikusan típusos funkcionális programozási nyelv. Mivel futási időben történő típushibák a program hibás működéséhez vezethetnek, ezért nagyon fontos a fordítási idejű minél pontosabb típusellenőrzés. Ilyen ellenőrzéseket az Erlang fordítóprogramja nem végez, ezért ehhez külső eszközökre van szükség. A RefactorErl egy az Eötvös Loránd Tudományegyetemen fejlesztett, Erlang nyelven írt programok elemzését és újrastrukturálását segítő keretrendszer. Dolgozatom célja, hogy bemutassak egy az Erlang nyelvre optimalizált típusellenőrző és függvényspecifikáció generáló eszközt, amely a RefactorErl-re épül. Ennek során ismertetem a kapcsolódó elméletet és annak megfeleltetését az Erlanghoz és a keretrendszerhez.

1. fejezet

Bevezetés

Egy jó programozási nyelv segíti a programozót, hogy a programot a feladatban szereplő fogalmakkal minél inkább hasonló fogalmakkal tudja leírni. Ez egyrészt megkönnyíti a munkáját, hiszen nincs szüksége a különböző absztrakciós szintek közötti kapcsolatok leírására, másrészt elősegíti, hogy az elkészült program az elvárt működést nyújtsa.

Az elvárt működés egyik előfeltétele, hogy a program az adatokat a nekik megfelelő módon kezelje, azaz az adatot a típusának megfelelő módon érjünk el, módosítsuk. Ennek ellenőrzését a fordítóprogram általában elvégzi. Amennyiben megsértjük a nyelv típusokra vonatkozó előírásait és ezt a fordítóprogram nem deríti fel, a programunk nagy valószínűséggel nem fog jól működni.

Az Erlang programozási nyelv fordítóprogramja nem végez ellenőrzéseket a típusokon. Az ellenőrzések a program futása közben történnek, ami hiba esetén a program abnormális működéséhez vezethet. Az ellenőrzést nem kell feltétlenül a fordítóprogramnak elvégeznie, hanem más eszközöket is igénybe vehetünk erre a célra. Ilyen például a TypEr vagy az Eötvös Loránd Tudományegyetemen fejlesztett RefactorErl [3] rendszerhez írt típuskikövetkeztető komponenes.

Dolgozatomban ezekről az eszközökről, a működésük alapelveiről fogok beszámolni. A következő kérdésekre keresek választ:

- *Mi a típuskikövetkeztetés?* Erre a kérdésre ad választ a 2. fejezet, amely-

ben bevezetem az elméleti alapokat, és bemutatok egy klasszikus módszert.

- *Miért fontos a típusbiztonság az Erlang programoknál?* A kérdésre a 3. fejezetben kapunk választ, amiben bemutatom az Erlang nyelvet.
- *Milyen eszközök léteznek a típushibák feltárására Erlang nyelvhez?* Erre a kérdésre a 3. fejezet végén adok választ, amiben bemutatok egy már létező eszközt.
- *Hogyan lehet megvalósítani egy típuskikövetkeztető rendszert?* A 4. és 5. fejezetekben részletesen ismertetek egy kifejezetten Erlanghoz készített típuskikövetkeztető rendszert. A 6. fejezetben bemutatom, hogy hogyan lehet egy típuskikövetkeztető rendszert integrálni a RefactorErl keretrendszerbe.

2. fejezet

Hindley-Milner algoritmus

Ebben a fejezetben összefoglalom a típuskikövetkeztetés elméleti alapjait. Bemutatok egy klasszikus tisztán funkcionális nyelvekhez használható algoritmust.

Elsőként vizsgáljuk meg, hogy az olyan klasszikus funkcionális nyelvek (például a Haskell [2] vagy a Clean [1]) milyen módszerek állnak rendelkezésre, hogy függvények és kifejezések típusát meghatározzuk. A funkcionális nyelvek alapja a λ -kalkulus [10]. Ez egy matematikai rendszer, amelyben a programokat szabályokkal írhatjuk le. Ennek az elméletét az 1930-40-es években fektették le, de igazi fénykorát a funkcionális nyelvek megjelenésével érte el. A λ -kalkulusnak két fő iránya van. A Church-típusos λ -kalkulus, ahol a kifejezéseknek leírásukkor megadott típusból explicit levezethető típusuk van. Ezzel szemben Curry-típusos λ -kalkulus esetén a kifejezéseknek nincs megadva típusa, de az kikövetkeztethető.

Ahhoz, hogy bemutassuk a Hindley-Milner algoritmus működését megadunk egy egyszerű, let-polimorf funkcionális nyelvet.

$$E ::= x \tag{2.1}$$

$$| E_1 E_2 \tag{2.2}$$

$$| \lambda x \rightarrow E \tag{2.3}$$

$$| \text{let } x = E_1 \text{ in } E_2 \tag{2.4}$$

Az első szabály a változókat jelenti. A második az applikáció¹ kifejezés. A harmadik az absztrakció kifejezése. A negyedik kifejezésben szereplő let polimorf, nem rekurzív let (értékadás kifejezés).

2.1. Típusrendszer

Az így definiált nyelvben nem szerepelnek típusok, de azokat ki lehet következtetni. A típusok egyrészt lehetnek típuskonstansok, mint az egész számok (*Int*), logikai értékek (*Bool*) vagy a karakterláncok (*String*). Lehetnek továbbá típusváltozók, vagy függvény (más forrásokban nyíl) típusok ($\tau_1 \rightarrow \tau_2$). Formálisan:

$$\tau ::= \alpha \mid Int \mid Bool \mid String \mid \tau_1 \rightarrow \tau_2$$

Típussémák. A típusok között lehetnek típussémák is. A típusséma $\forall \vec{\alpha}. \tau$ alakú típuskifejezés, ahol $\vec{\alpha} = \alpha_1, \dots, \alpha_n$. Egy típussémában a $\forall$ kvantorok kötik a típusváltozókat, és a típussémában csak univerzális kvantorok által kötött típusváltozók vannak. A típussémában szereplő típusváltozók sorrendje nem lényeges.

Szabad változók. Egy τ kifejezés típusában szereplő szabad változókat $FV(\tau)$ jelöli és a kifejezés változóit tartalmazza. A típusséma szabad változói a sémában nem szereplő típusváltozókat tartalmazza, azaz

$$FV(\forall \vec{\alpha}. \tau) = FV(\tau) \setminus \vec{\alpha}.$$

Helyettesítés. A típusokban van lehetőség a típusváltozókat típusokkal helyettesíteni (jelölje a leképezést S). A konkrét helyettesítést az $[\alpha := \tau]$ szimbólumsorozattal jelöljük. Amennyiben több változót szeretnénk egy lépésben helyettesíteni, akkor az $[\alpha_1 := \tau_1, \dots, \alpha_n := \tau_n]$ szimbólumsorozattal jelöljük. A helyettesítés érintetlenül hagy minden olyan típusválto-

¹Mivel nincsenek függvények, ezért függvényhívásról nem beszélhetünk.

zót a típusban, ami nem szerepel az S értelemezési tartományában. Továbbá a helyettesítés csak a nem kötött típusváltozókat érinti.

Típuskörnyezet. A típuskörnyezet egy olyan függvény, amelyben a változókhoz típusokat és típussémákat rendelünk. A típus tartalmazhat típussémát is. Szokások jelölése: Γ . A típuskörnyezet szolgáltatja a típuskikövetkeztetés kontextusát. A típuskörnyezetből (Γ) eltávolíthatunk egy változót (x), ekkor a keletkezett típuskörnyezetet $\Gamma \setminus x$ -szel jelöljük. Beszélhetünk egy típuskörnyezet szabad változóiról, ami a típuskörnyezetben található típusok és típussémák szabad változóinak halmazát jelenti. Továbbá definiáljuk a típuskörnyezet egy helyettesítését is úgy, hogy az S helyettesítést a típuskörnyezetben elemenként végezzük.

Típus általánosítása. Bevezetjük egy τ típus általánosítását a Γ típuskörnyezetre nézve. Az általánosítás az a típus $gen(\Gamma, \tau) = \forall \vec{\alpha}. \tau$, ahol $\vec{\alpha} = FV(\tau) \setminus FV(\Gamma)$. Tehát a τ egyszerű típus általánosításához csak azon szabad változóira kell a $\forall$ kvantorral kötést adni, amelynek nincsenek benne a Γ szabad változói között.

Példányosítás. Egy típusséma konkrét típusát, azaz egy példányát a példányosítás műveletével kapjuk meg. Egyszerű típus (τ) esetén

$$inst(\tau) = \tau,$$

míg típusséma esetén, ha $\sigma \equiv \forall \alpha_1 \dots \alpha_n. \tau$, akkor

$$inst(\sigma) = \tau[\alpha_1 := \tau_1, \dots, \alpha_n := \tau_n],$$

ahol $\tau_1, \dots, \tau_n$ új típusváltozókat tartalmazó típusok. A $inst(\sigma)$ típust a σ egy generikus példányának nevezzük.

Legáltalánosabb egyesítő helyettesítés. A τ_1 és τ_2 típuskifejezések legáltalánosabb egységesítő helyettesítése S , ha $S\tau_1 \equiv S\tau_2$, azaz egységesítő

helyettesítés, és minden S_1, S_2 egységesítő helyettesítésre van olyan S helyettesítés, hogy $S_1 \equiv SS_1$. Jelölje ezt a helyettesítést $mgu(\tau_1, \tau_2)$.

2.2. Típusszabályok

A következőkben ismertetjük azokat a következtetési szabályokat, amelyeket a korábban ismertetett nyelvi konstrukcióhoz adhatunk. A $\Gamma \vdash e : \tau$ következtetést úgy olvassuk, hogy a Γ típuskörnyezetből levezethető az e kifejezésre a τ típus. Ez egy szintaxis vezérelt módszer, mivel egy kifejezés szintaktikája határozza meg a következtetés lépéseit.

Változó kifejezés

A változó kifejezés típusa τ , feltéve, hogy a típuskörnyezet tartalmazza az x változóhoz a τ típust.

$$\frac{\Gamma \cup \{x : \tau\} \vdash \tau}{\Gamma \vdash x : \tau} \quad (\text{Var})$$

Applikáció kifejezés

Amennyiben az e_1 nyíl típusú, és az e_2 típusa a nyíl típus paraméter típusának felel meg, akkor az applikáció típusozható, és a típusa a nyíl típus visszatérési értéke.

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad (\text{App})$$

Absztrakció kifejezés

Az absztrakció csak akkor típusozható, ha az absztrahált változó nincs benne a típuskörnyezetben. A kifejezés típusa nyíl típus.

$$\frac{\Gamma \setminus x \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x \rightarrow e : \tau_1 \rightarrow \tau_2} \quad (\text{Abs})$$

Polimorf let kifejezés

A polimorf let kifejezés típusozásához van szükségünk az általánosított típuséma bevezetésére.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \setminus x \cup \{x : \text{gen}(\Gamma, \tau_1)\} \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (\text{Let})$$

Így elérhetjük azt, hogy a típusémák csak a típuskörnyezetben szerepeljenek, és azt, hogy az egyszerű típus általánosítása csak a let kifejezésben szereplejen.

2.3. A W algoritmus

Számos olyan algoritmust leírtak, amely egy kifejezés típusát határozza meg típuskörnyezetek felhasználásával. Az egyik legismertebb ilyen algoritmus az úgynevezett W algoritmus. Ez egy alulról felfelé elemző szintaxis vezérelt algoritmus, amelyben a példányosítás és az általánosítás művelete explicit módon van megadva. A W algoritmus egy adott típuskörnyezethez meghatároz egy helyettesítést és egy típust, vagy *fail* hibajelzéssel leáll, azaz

$$W : \Gamma \times E \rightarrow \begin{cases} S \times \tau \\ \text{fail} \end{cases}$$

Az algoritmust az E kifejezés szerkezete szerint adjuk meg és a 2.1. ábra mutatja. Az algoritmusban szereplő U függvény a paraméteréül kapott típusegyenletek megoldását adja. Amennyiben a $W(\Gamma, E)$ eredménye az (S, τ) , akkor az E kifejezés eredményét a $S\Gamma \vdash E : \tau$.

$$\begin{aligned}
W(\Gamma, x) &= (Id, inst(\sigma)), \text{ ahol } \{x, \sigma\} \in \Gamma \\
W(\Gamma, \lambda x \rightarrow E) &= (S, S\alpha \rightarrow \tau), \text{ ahol} \\
&\quad W(\Gamma \cup \{x : \alpha\}, E) = (S, \tau), \alpha \text{ új típusváltozó} \\
W(\Gamma, EF) &= (S_1 S_2 S_3, S_3 \alpha), \text{ ahol} \\
&\quad W(\Gamma, E) = (S_1, \tau') \\
&\quad W(S_1 \Gamma, F) = (S_2, \tau'') \\
&\quad S_3 = U(S_2 \tau', \tau'' \rightarrow \alpha), \alpha \text{ új típusváltozó} \\
W(\Gamma, \text{let } x = E \text{ in } F) &= (S_1 S_2, \tau''), \text{ ahol} \\
&\quad W(\Gamma, E) = (S_1, \tau') \\
&\quad W(S_1 \Gamma \cup \{x : gen(S_1 \Gamma, \tau')\}, F) = (S_2, \tau'').
\end{aligned}$$

2.1. ábra. A W algoritmus.

3. fejezet

Erlang, mint funkcionális nyelv

Az Erlang [5], [8] programozási nyelvet az Ericsson fejlesztette ki azzal a céllal, hogy telekommunikációs eszközök programozási nyelve legyen, melyeknél a fő szempont a nagy forgalomáteresztő képesség, a párhuzamosság és elosztottság kimagasló támogatása nagy hibatűrés mellett. Az ilyen rendszerek fejlesztésének rendkívül magas erőforrásigénye szükségessé tette, hogy az új nyelv könnyen elsajátítható legyen. Miután több 80-as években létező programozási nyelvet megvizsgáltak, hogy alkalmasak-e a feladatra, azt állapították meg, hogy a Prolog deklaratív stílusa volt a leginkább alkalmasabb arra, hogy ilyen rendszereket fejlesszenek. Így az Erlang Prolog metanyelvként kezdte, majd abból alakult ki sajátos stílusa.

Az Erlang nyelvet a funkcionális nyelvek közé sorolják, de a fenti tervezési irányelvek mentén több olyan nyelvi elemet tartalmaz, ami nem jellemző más ma használt funkcionális nyelvre (Clean, Haskell). Ilyen például, hogy a függvények különböző végrehajtási ágai más-más típusú értékkel térhetnek vissza, ugyanez igaz az egyes paraméterekre is. Továbbá lényeges különbség, hogy bár vannak benne függvény objektumok¹, a függvények paraméterei nem curry-zhetők. Megjelenik a nyelvben az imperatív programozási nyelvek legfőbb eleme a szekvencia is. Ugyanakkor megtalálható benne több olyan elem is, ami mégis azt mutatja, hogy az Erlang inkább funkcionális nyelv. Ilyen a helyfüggetlen változó bevezetése,

¹ Az objektum szót entitás értelemben fogom használni.

illetve a függvények deklaratív stílusa.

Nagy rendszerek tervezésénél nagy jelentősége van, hogy a felhasznált nyelv mennyire támogatja a programozási hibák minél hamarabb történő felfedezését. Ilyen hibák lehetnek a szintaktikus hibák, de a fordító program sok egyéb szemantikus hibát is észlelhet (mint például a változók kötésének ellenőrzése). Az egyik ilyen klasszikus fordítási ellenőrzés a program típushelyessége. Egy jól megtervezett programban a típushiba általában a programozó hibájára utal, ugyanakkor felfedhet tervezési hibákat is. A fordítási időben észlelt hibák kijavítása lényegesen kevesebb erőforrást igényel, mintha ugyanazt a hibát teszteléssel lehet csak felfedezni.

Erlangban csak a nyelv saját típusai léteznek, a felhasználó nem definiálhat új típust. Azonban van lehetőség az elemi típusokból bonyolultabb adatszerkezetek építésére, de a fordítóprogram nem ellenőrzi ezen szabályok betartását. Ez vállalható kompromisszumot jelentett egyrészt a fordító program bonyolultsága másrészt a nyelv használhatósága között.

A programozói hibák hatásainak korlátozására több módszer is van Erlangban. A nyelv mellé kifejlesztettek egy robusztus függvénykönyvtárat, amelyben több tervmintát is megvalósítottak. Ezek egyike a felügyelő minta, ami az Erlang által támogatott párhuzamos folyamatok (processzek) felügyelését teszi lehetővé. Mivel a processzek kis erőforrás igényűek (például könnyű létrehozni őket), ezért célszerű sok apró folyamatot implementálni. Egy process abortálása a programozó által megadott módon van hatással a program többi részére, ezért a processzen belüli hiba hatása jól lokalizálható.

3.1. A Dialyzer

Azáltal, hogy a programokban előforduló hibák hatása kis helyre koncentráldódik, még mindig nehéz megtalálni azokat. Az „elkészült” programot minden lehetséges futási ágon tesztelni nehéz és költséges folyamat. Ez a feladat még nehezebb, ha párhuzamosság is van a programban. Éppen ezért kulcsfontosságú volt az olyan eszközök kifejlesztése, amik képesek

megtalálni a program futtatása nélkül a hibákat. Erlanghoz a legfontosabb ilyen eszköz a Dialyzer, amelyet a HiPe csoport fejlesztett ki Tobias Lindahl és Konstantinos Sagonas vezetésével. Ez a rendszer a lefordított Erlang bájtkódból előállított magnyelvet elemzi és olyan pontokat keres a programban, ahol biztosan hiba lépne fel ha futtatnánk a programot. A Dialyzert úgy tervezték, hogy a lehető legtöbb hibát megtalálja a programban, viszont ne jelezzon fals-pozitív hibákat. Továbbá nem igényel semmilyen változtatást a nyelvben, így jól használható már megírt alkalmazásokhoz is. További fontos szempont, hogy akkor is a lehetőségekhez képest megbízható eredményt kell produkálnia, ha a program egyes részei nem elérhetőek. Ezeket a tervezési szempontokat több más Erlanghoz írt rendszer is magáénak tekintette. A Dialyzer sikerességét jelzi, hogy azóta része lett az Erlang függvény és alkalmazáskönyvtárának (OTP – Open Telecom Platform).

A Dialyzer statikus analízist végez a lefordított bájtkódon, aminek az eredményét eltárolja, hogy az később újrafelhasználható legyen. Az eltárolt információk között megtalálható egy függvényhívási gráf, az egyes kifejezések típusa az egyes végrehajtási útvonalak mentén. Ezen információk alapján az egyes függvényekhez lehet generálni specifikációt. Az, hogy ez a specifikáció milyen formát ölt, nagyon fontos a felhasználónak. Ha ugyanis minden végrehajtási úthoz meghatározzuk a típust, és ezt a felhasználó rendelkezésére bocsátjuk, akkor az – bár pontos – nem segíti, hanem megbonyolítja a felhasználó életét. Ilyen feltételes típusokkal és következtetésükkel foglalkozik a következő cikk [4]. Klasszikus funkcionális nyelveknél feltételek nem kellenek, mert a függvények különböző ágai ugyanolyan típusú adattal kell visszatérjenek.

Egy másik módja a felhasználó számára könnyen értelmezhető specifikáció kinyerésének, hogy elhagyjuk a feltételeket a típusinformációból. Ezt a módszert követi a Dialyzerre épülő TypEr. Ennek működési elvéről szól a 4. fejezet.

4. fejezet

Success typing

A TypEr Erlang függvények specifikációinak generálására szolgál. A megvalósítása a Dialyzerre épül, és a hagyományos (Hindley-Milner-hez hasonló) típuskikövetkeztető módszertől eltérően egy új algoritmus alapján határozza meg a függvények típusát. Az algoritmust success typing-nak¹ nevezik. A fejezetben ezt az algoritmust ismertetem Tobias Lindahl és Konstantinos Sagonas cikke alapján. [6]

A funkcionális nyelven programozók gyakran szembesülnek azzal, hogy a típusrendszer szigorúsága miatt bonyolultabb programokat kell írjanak. Nem vezethetnek be a program egy adott pontján a típushoz képest extrémális elemet, anélkül, hogy ne definiálna hozzá egy új típust. Bár az új típussal nő a program megbízhatósága, ugyanakkor nagy, produktív rendszerek fejlesztésénél ez nagy többletterhet ró a programozóra. Erlangban ezzel szemben a programozót nem kötik ilyen szabályok, amit gyakran ki is használnak.

Az Erlang dinamikusan típusos programozási nyelv, ugyanakkor szigorú típusrendszere van. Ez azt jelenti, hogy a típusok között nincs a fordítóprogram által automatikusan elvégzett konverzió. A success typing erősen kihasználja az Erlang beépített típusrendszerét.

¹Mivel a magyar szakirodalomban nincs fordítása, ezért a továbbiakban az angol kifejezést fogom használni.

4.1. Altípusrendszerek

Amíg a hagyományos típuskikövetkeztető rendszerek egyesítő eljárásai $\alpha = \beta$ alakú típusegyenletek megoldásával foglalkozik, addig az altípusosságon alapuló rendszerek $\alpha \subset \beta$ alakú egyenletekkel. Ezáltal még általánosabbak, az előzőnél.

Miért van szükség erre az általánosságra? A kérdés megválaszolásához meg kell vizsgálni, hogy mit is jelent a gyakorlatban az, hogy a függvénynek több típusú visszatérési értéke lehet. Ehhez tekintsük a következő függvény definíciót:

```
fact(N) when is_integer(N) and (N>0)->
    N * fact(N - 1);
fact(0) ->
    1;
fact(N) when is_integer(N) ->
    negative;
fact(_) ->
    error.
```

Ez az egyszerű implementációja a faktoriális függvénynek, jól példázza az Erlang egyszerűségét. Minden nemnegatív egész számra visszaadja a szám faktoriálisát, aminek típusa egész szám (integer), míg bármilyen más egész értékre a `negative` atommal tér vissza, bármilyen más típusú értékre az `error` atommal. Az ilyen függvények típusozásával a Hindley-Milner-féle módszerek nem boldogulnak.

Más funkcionális nyelvekben jártas programozóknak ez szokatlannak vagy kerülendőnek tűnik, azonban valódi Erlang programok nagyon gyakran használnak ilyen és ehhez hasonló technikákat. Éppen ezért egy ipari használatra tervezett típuskikövetkeztetőnek kezelnie kell az ilyen eseteket.

A probléma feloldására bevezetjük az altípusosságot és a fenti példában található esetekben megengedjük az unió típust, mint visszatérési értéket. Az unió típust használjuk a paraméterek esetében is. Ez esetben már

meghatározhatjuk az előző példánál, hogy a függvény tetszőleges típusú értékkel meghívva szám vagy atom típusú értéket ad vissza.

Meg kell jegyezni, hogy az altípusosság alkalmazása nem ismeretlen dinamikusan típusos nyelvek esetében, még Erlang esetében sem. Marlow és Wadler [9] már javasoltak egy altípusosságon alapuló típuskikövetkeztető rendszert. Azonban a mintaillesztés nehézségeket jelentett, amit a következő példán szemléltettek²:

```
and(true, true) -> true;
and(false, _)   -> false;
and(_, false)  -> false.
```

Jelöljük az összes Erlang termet *any()*-vel, a `true` $\cup$ `false` unió típust *bool()*-lal. Ekkor a Hindley-Milner algoritmus $(bool(), bool()) \rightarrow bool()$ típust határozza meg a fenti függvényhez. Figyelembe véve a *bool()* definícióját a típus helyesnek tekinthető, ha feltesszük, hogy az atomok nem részei más típusértékalmazoknak. Ugyanakkor az Erlang nyelv szemantikája nem tiltja, hogy a függvényt a következő paraméterekkel hívjuk meg `and(false, 42)`. Ez nem fog típushibát okozni, bár nem valószínű, hogy az eredeti függvény írója gondolt erre a használati esetre, ezt sem lehet elutasítani.

Marlow és Wadler által javasolt algoritmus ugyanakkor a $(any(), false) \rightarrow bool()$ típust következteti ki. Ez nem altípusa a Hindley-Milner által generált típusnak, ráadásul egyáltalán nem fejezi ki a függvény írójának szándékát. Ez a típuskikövetkeztető rendszer nem terjedt el az Erlang programozók körében.

A success typing célkitűzései. Az új típuskikövetkeztető rendszer céljai részben azonosak a Dialyzerével, azaz egy minél pontosabb, fals-pozitív hibáktól mentes, emberi beavatkozás nélkül működő eszköz készítése. A már megírt produktív rendszerek esetében a programkód kiegészítése típusinformációkkal rendkívül költséges, sőt néha nem engedélyezett, ezért

²A példa az általuk megjelentetett cikk 9.3-as szakaszában található.

a típuskikövetkeztetés nem alapulhat a nyelv és a kód módosításán. Továbbá akkor is elfogadható eredményt kell adjon, amikor a programkód egy része hiányzik. Ilyen eset például akkor fordulhat elő, ha a program egyes részei egy másik programozási nyelven (például C) lettek implementálva, vagy egyes szoftverkomponensek forrása nem elérhető licencelési okok miatt.

4.2. Az Erlang magnyelv

Az algoritmus szerkezetének és logikájának egyszerű tárgyalásához szükséges az Erlang nyelvnek megragadni azokat a részeit, amelyek lényegében befolyásolják a nyelv szemantikáját. Ezért az algoritmust az Erlang magnyelvén³ tárgyaljuk. Ez az a köztes nyelv, amely tartalmazza az Erlang nyelv teljesen funkcionális részét megőrizve a szemantikáját. Az Erlang magnyelv szintaktikája a 4.1. ábrán látható. A nagybetűk (X) változókat jelölnek, az indexelt kisbetűk a nekik megfelelő szabályt.

Az egyes szabályok operációs szemantikáját a kikövetkeztetési szabályok indoklásánál ismertetem.

$$\begin{aligned}
e &::= X \mid c(e_1, \dots, e_n) \mid e_1(e_2, \dots, e_n) \mid f \mid \text{let } x = e_1 \text{ in } e_2 \mid \\
&\quad \text{letrec } x_1 = f_1, \dots, x_n = f_n \text{ in } e \mid \\
&\quad \text{case } e \text{ of } (p_1 \rightarrow b_1); \dots; (p_n \rightarrow b_n) \text{ end} \\
f &::= \text{fun } (x_1, \dots, x_n) \rightarrow e \\
p &::= p' \text{ when } g \\
p' &::= x \mid c(p'_1, \dots, p'_n) \\
g &::= g_1 \text{ and } g_2 \mid x_1 = x_2 \mid \text{true} \mid \text{is_atom}(x) \mid \text{is_integer}(x) \mid \dots
\end{aligned}$$

4.1. ábra. Az Erlang magnyelv

A teljes Erlang nyelv típusait a 4.2. ábra tartalmazza. A mintaillesztés szükségessé teszi, hogy a típusérték-halmazok elemeit külön típusként

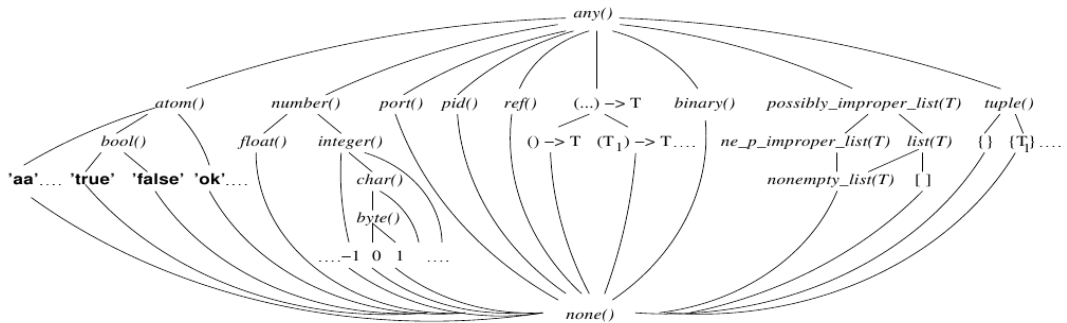
³Core Erlang

$$\begin{aligned}
T &::= A \mid I \mid \text{float}() \mid C \mid \text{binary}() \mid S \mid L \mid F \mid (T_1 \cup T_2) \mid \text{any}() \mid \\
&\quad \text{none}() \mid c(T_1, \dots, T_n) \mid V, \\
A &::= '' \mid 'a' \mid 'b' \mid \dots \mid 'aa' \mid \dots \mid 'ok' \mid 'true' \mid 'false' \mid \dots \mid \text{atom}() \\
I &::= \dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots \mid \text{byte}() \mid \text{char}() \mid \text{integer}() \\
C &::= \text{pid}() \mid \text{port}() \mid \text{ref}() \\
S &::= \{T_1, \dots, T_n\}, n \geq 0 \mid \text{tuple}() \\
L &::= [] \mid \text{list}(T) \mid \text{nonempty_list}(T) \mid \text{possibly_improper_list}() \mid \\
&\quad \text{nonempty_possibly_improper_list}(T) \\
F &::= (T_1, \dots, T_n) \rightarrow T, n \geq 0 \mid (\dots) \rightarrow T \\
V &::= \alpha, \beta, \gamma, \tau, \dots
\end{aligned}$$

4.2. ábra. Az Erlang magnyelv típusai

kezeljük. Ugyanis például az $f(ok) \rightarrow ok$. definícióval értelmezett függvény meghívása bármilyen más értékkel, mint az ok atom típushibához vezet. Az $\text{any}()$ típus az összes Erlang termet magában foglalja. A $\text{none}()$ típus típusértékhalmaza üres, ez a típus jelzi a típushibát.

A típusok közötti altípus-relációt a 4.3. ábra definiálja úgy, hogy az élekkel összekötött típusok relációban állnak egymással oly módon, hogy a gyerek csúcsban lévő típus altípusa a szülő csúcsban lévőknek. Az unió típust a $\cup$ szimbólummal jelöljük.



4.3. ábra. Az Erlang magnyelv altípusrendszere [7]

Az Erlang magnyelvben az egyszerű (egészek, atomok stb.) és az ezek-

ből képzett összetett típusokat $c(e_1, \dots, e_n)$ kifejezés jelentette, így a T szabályt kiegészítjük a $c(T_1, \dots, T_n)$ szabállyal. Továbbá megengedjük a típusok mellett a típusváltozókat is, melyeket görög betűkkel $(\alpha, \beta, \gamma, \dots)$ jelölünk.

4.3. Success typing fogalma

Az alapötlet a success typing mögött az, hogy megpróbáljuk meghatározni azt a legtágabb típusértékalmazt, amire biztosak lehetünk, hogy alkalmazásuk esetén típushiba lép fel. Mindezt megtehetjük, hiszen szemben a statikus típusrendszerrel rendelkező nyelvekkel, nem akarunk típusbiztosságot bizonyítani, sőt a dinamikus típusellenőrzést sem szeretnénk lecserélni. Ahelyett, hogy egy olyan algoritmust terveznénk, ami kiszámolja a fenti halmazt, éppen az ellenkezőjét tesszük, kikövetkeztetjük a komplementerét. Az így keletkezett típust a függvény „sikeres típusának”⁴ (angolul success typing) nevezzük.

Formálisan az f függvény esetén a success typing az a típusspecifikáció $((\alpha_1, \dots, \alpha_n) \rightarrow \beta)$, amire igaz, hogy valahányszor az $f(p_1, \dots, p_n)$ függvényhívás értéke v , akkor $v \subset \beta$ és $\forall i \in [1..n] : p_i \subset \alpha_i$

Fontos megjegyezni, hogy a statikus típusrendszer és a success typing típusspecifikációja között nagy különbség van. Az utóbbi megmutatja a függvény összes szándékos használati módját és *még néhányat*. Ez azt jelenti, hogy ha nem a specifikáció szerint alkalmazzuk a függvényt, akkor biztosan típusütközés lesz.

4.4. Korlátozás generálás és operációs szemantika

A success typing algoritmus is korlátozások generálásán és azok megoldásán alapszik. A következőkben ismertetem az Erlang magnyelv kifejezéseihez tartozó korlátozásokat és egyben megadom az operációs szemantikát is. A szabályok megadásánál A típuskörnyezetet jelent, amelyben a

⁴A fordítást nem használjuk a továbbiakban.

változók és a nekik megfeleltetett típus található. Ennek formája

$$\{\dots, x \mapsto \tau_x, \dots\}.$$

A korlátozáshalmaz altípus relációk, és korlátozások egymásba ágyazott konjunkciója és diszjunkciója.

$$C ::= (T_1 \subset T_n) \mid (C_1 \wedge \dots \wedge C_n) \mid (C_1 \vee \dots \vee C_n)$$

Két típus közötti egyenlőség esetén használjuk a $T_1 = T_2$ jelölést, ami megfelel a $(T_1 \subset T_2) \wedge (T_2 \subset T_1)$ alakú kifejezésnek. Használni fogjuk a következő jelölést

$$A \vdash e : \tau, C,$$

amit úgy olvasunk ki, hogy „adott A típuskörnyezet esetén az e kifejezés típusa τ a C korlátozáshalmazzal”.

Változók

$$\frac{}{A \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset} \quad [\text{Var}]$$

A változók esetében bővítjük a típuskörnyezetet, és nem generálunk új korlátozást.

Konstruktor

$$\frac{A \vdash e_1 : \tau_1, C_1 \dots e_n : \tau_n, C_n}{A \vdash c(e_1, \dots, e_n) : c(\tau_1, \dots, \tau_n), C_1 \wedge \dots \wedge C_n} \quad [\text{Struct}]$$

A c operátor a paraméterként kapott kifejezésekből összetett kifejezéseket generál. Ezzel szimuláljuk az egyszerű és összetett típusokat is. A [Struct] szabály azt fejezi ki, hogy ha tudjuk az alap kifejezés típusait, akkor a c konstruktort alkalmazva rájuk a kifejezés típusa is a konstruktor típusának megfelelő típus lesz. Az alap kifejezésekhez tartozó korlátozáshalmazoknak a konjunkcióját kell venni, hiszen minden korlátozásnak teljesülnie kell.

Egyszerű let kifejezés

$$\frac{A \vdash e_1 : \tau_1, C_1 \quad A \cup \{x \mapsto \tau_1\} \vdash e_2 : \tau_2, C_2}{A \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2, C_1 \wedge C_2} \quad [\text{Let}]$$

Az egyszerű let kifejezés nem szerepel az Erlang nyelvben, de a mag nyelv része, ugyanis ez használható függvények szimulálására. Az e_1 kifejezést típusát meghatározva bővítjük a típuskörnyezetet ezzel a típusal, majd így meghatározzuk a e_2 típusát. Ennek birtokában a let kifejezés típusa e_2 típusával egyezik meg. A keletkezett korlátozáshalmazoknak a konjunkcióját vesszük.

Rekurzív let kifejezés

$$\frac{A \cup \{x_i \mapsto \tau_i \mid i \in [1..n]\} \vdash f_1 : \tau'_1, C_1 \dots f_n : \tau'_n, C_n \quad e : \tau, C}{A \vdash \text{letrec } x_1 = f_1, \dots, x_n = f_n \text{ in } e : \tau, C_1 \wedge \dots \wedge C_n \wedge \wedge (\tau'_1 = \tau_1) \wedge \dots \wedge (\tau'_n = \tau_n)} \quad [\text{Letrec}]$$

A rekurzív let kifejezést megkülönböztetjük az egyszerű let kifejezéstől, mivel a típuskörnyezetben másképp kell kezelni a rekurzív kifejezéseket. A típuskörnyezetet bővítjük rekurzív típusváltozókkal, amelyek a hatóköre kiterjed mind a függvény kifejezésekre, mind a letrec kifejezés törzsére is. A megszorítás halmazok diszjunkciójához további megszorításként hozzávesszük, hogy a rekurzív típusváltozóknak meg kell egyezzenek az egyes függvények számított típusával is.

Függvény alkalmazás

$$\frac{A \vdash e_1 : \tau_1, C_1 \dots e_n : \tau_n, C_n}{A \vdash e_1(e_2, \dots, e_n) : \beta, (\tau_1 = (\alpha_2, \dots, \alpha_n) \rightarrow \alpha) \wedge (\beta \subseteq \alpha) \wedge \wedge (\tau_2 \subseteq \alpha_2) \wedge \dots \wedge (\tau_n \subseteq \alpha_n) \wedge C_1 \wedge \dots \wedge C_n} \quad [\text{App}]$$

A függvényhívás kissé eltér a szokásostól. A visszatérési érték típusa a hagyományos altípusrendszerekben kontravariánsan viszonyul a függvény alkalmazásbélihez. Ezzel a szabállyal a típusbiztonságot háttérbe helyezzük a sokkal fontosabb a fals-pozitív típusütközések elkerülésénél. Intuitívan abból indulunk ki, hogy ha a függvényhívás sikeres, akkor a

visszaadott érték altípusa lesz az eredeti függvény értékkészletének. Az aktuális paraméterek típusának természetesen kovariánsan kell viszonyuljon az eredeti függvény paramétereinek típusához. A korlátozásoknak teljesülniük kell minden esetben, ezért a konjukciójukat vesszük.

Absztrakció

$$\frac{A \cup \{x_i \mapsto \tau_i \mid i \in [1..n]\} \vdash e : \tau_e, C}{A \vdash \text{fun}(x_1, \dots, x_n) \rightarrow e : \tau, (\tau = ((\tau_1, \dots, \tau_n) \rightarrow \tau_e)) \wedge C} \quad [\text{Abs}]$$

Az Erlang nyelvben a függvények nem curry-zhetők, emiatt a típusuk egy olyan függvény típus, ahol a paraméterek száma és a visszatérési érték típusa adott. Függvények esetében a típuskörnyezetet itt is bővítjük a paramétereknek megfelelő típusváltozóval, így azok hatóköre kiterjed a teljes függvény kifejezésre.

Minta őrfeltétellel

$$\frac{A \vdash p : \tau, C_p \quad A \vdash g : \text{true}, C_g}{A \vdash p \text{ when } g : \tau, C_p \wedge C_g} \quad [\text{Pat}]$$

Az Erlang nyelv nagyon fontos része a mintaillesztés. A magnyelvben a mintáknak mindig őrfeltétellel kell álljanak. Az őrfeltételeknek igazra kell kiértékelődniük, különben típusütközés történik. Az őrfeltétel nem csak egy egyszerű kifejezés lehet, hanem több kifejezés konjukciója vagy diszjunkciója.

case kifejezés

$$\frac{\begin{array}{c} A \cup \{v \mapsto \tau_v \mid v \in \text{Var}(p_1)\} \vdash p_1 : \alpha_1, C_1^p, b_1 : \beta_1, C_1^b \\ \vdots \\ A \cup \{v \mapsto \tau_v \mid v \in \text{Var}(p_n)\} \vdash p_n : \alpha_n, C_n^p, b_n : \beta_n, C_n^b \end{array}}{A \vdash \text{case } e \text{ of } p_1 \rightarrow b_1; \dots p_n \rightarrow b_n \text{ end} : \beta, C_e \wedge (C_1 \vee \dots \vee C_n) \text{ ahol } C_i = ((\beta = \beta_i) \wedge (\tau_i = \alpha_i) \wedge C_i^p \wedge C_i^b)} \quad [\text{Case}]$$

A case kifejezés az egyik legösszetettebb a magnyelvben, ugyanis az eredeti nyelv több szerkezetét is lehet vele szimulálni. A kifejezés egyetlen

ágának (klózának) sikeres típusozása elég ahhoz, hogy az egész kifejezés típusozása sikeres legyen. Éppen ezért itt bevezetjük a korlátozások diszjunkcióját. Minden a diszjunkcióban lévő korlátozás tartalmazza az összes olyan korlátozást, amelynek teljesülnie kell ahhoz, hogy a neki megfelelő klóz típusozható legyen. Az egyes klózoknál külön-külön bővítjük az eredeti típuskörnyezetet a mintákban szereplő változókhoz tartozó típusváltozókkal, így az adott klózban szereplő korlátozások hatása lokális lesz az adott klózra. A *Var* függvény visszaadja a mintában szereplő kötetlen változókat.

Fontos, hogy a korlátozások generálásánál nem kötöttük ki, hogy a case kifejezés visszatérési értékére, hogy annak unió típussal kell visszatérnie. Ezt elég megtenni a korlátozások interpretálásánál, tehát az egyesítő algoritmusnál.

4.5. Az egyesítő algoritmus

Az egyesítő algoritmust használjuk arra, hogy a korlátozáshalmazokat feldolgozva meghatározzuk a kifejezések típusát. Ehhez szükségünk lesz egy leképezésre, ami típuskifejezésekről és típusváltozókról képez konkrét típusokra. Jelölje ezt a leképezést *Sol*. Továbbá az *S* megoldása a *C* korlátozáshalmaznak ($S \models C$), ha

$$\begin{aligned} S \models T_1 \subset T_2 &\iff \text{none}() \subset \text{Sol}(T_1) \subset \text{Sol}(T_2) \\ S \models C_1 \wedge C_2 &\iff S \models C_1 \wedge S \models C_2 \\ S \models C_1 \vee C_2 &\iff S_1 \models C_1 \wedge S_2 \models C_2 \wedge S = S_1 \sqcup S_2, \end{aligned}$$

ahol $S_1 \sqcup S_2$ jelenti a pontonkénti felső határát a megoldás halmaznak. Ez praktikusán azt jelenti, hogy a két halmazban közös típusoknak képezzük az unióját. Ha egy altípus-reláció nem teljesül, akkor a megoldás az üres megoldás lesz, amit $\perp$ jelöl. Ebben a megoldáshalmazban minden kifejezéshez és típusváltozóhoz a *none()* típus tartozik. Megjegyezzük, hogy $\perp \sqcup S = S$, ami éppen a case kifejezésnek megfelelő viselkedését írja le.

A megoldó algoritmust a 4.4. ábra tartalmazza. A `solve` függvény fel dolgozza a korlátozáshalmazt, és módosítja a megoldáshalmazt. A megoldáshalmaz kezdetben a korlátozásgeneráláskor felhasznált típusváltozókat tartalmazza, a hozzájuk tartozó típus pedig az univerzális típus, az `any()`. A rekurzív típusváltozók kivételt képeznek ez alól, mivel hozzájuk a `none()` típus tartozik. Erre azért van szükség, hogy a típusváltozóhoz tartozó típus megváltozzon az algoritmus egy iterációja során. Ezáltal az algoritmus újra fog futni immár az új megoldáshalmazzal.

Az algoritmus akkor ér véget, ha a `solve` függvény nem változtat a megoldáshalmazon, azaz az algoritmus fixpontba jutott. Amennyiben a rekurzív típusváltozóhoz sikerült a `none()`-tól eltérő típust rendelni, akkor az algoritmus újra lefut, de már ezzel a módosított értékkel. Az algoritmus terminálását az garantálja, hogy a bővülő típusérték-halmazok egy pont után elérik az univerzális típust, ami tovább nem bővíthető. Ez természetesen azzal jár, hogy az unió típus képzésénél amennyiben az unió elemszáma elér egy bizonyos határt, akkor konkrét típusokat az őket befoglaló általánosabb típussal helyettesítjük. Például, amennyiben az unió elemszámának felső korlátja öt, akkor az $1 \cup 2 \cup 3 \cup 4 \cup 5$ unió típust az `integer()` egész számokat tartalmazó típussá kerekítjük.

$$\begin{aligned}
\text{solve}(\perp, _) &= \perp \\
\text{solve}(\text{Sol}, \alpha \subseteq \beta) &= \begin{cases} \text{Sol} & : \text{Sol}(\alpha) \subseteq \text{Sol}(\beta) \\ \text{Sol}[\alpha \mapsto T] & : T = \text{Sol}(\alpha) \sqcap \text{Sol}(\beta) \neq \text{none}() \\ \perp & : T = \text{Sol}(\alpha) \sqcap \text{Sol}(\beta) = \text{none}() \end{cases} \\
\text{solve}(\text{Sol}, \text{Conj}) &= \begin{cases} \text{Sol} & : \text{solve_conj}(\text{Sol}, \text{Conj}) = \text{Sol} \\ \text{solve}(\text{Sol}', \text{Conj}) & : \text{Sol}' = \text{solve_conj}(\text{Sol}, \text{Conj}) \neq \text{Sol} \end{cases} \\
\text{solve}(\text{Sol}, \text{Disj}) &= \begin{cases} \bigsqcup \text{Sol}' & : \text{Sol}' \neq \emptyset, \text{ ahol } \begin{cases} \text{Sol}' = \{S \mid S \in \text{PS}, S \neq \perp\} \\ \text{PS} = \{\text{solve}(\text{Sol}, C) \mid C \in \text{Disj}\} \end{cases} \\ \perp & : \text{Sol}' = \emptyset \end{cases}
\end{aligned}$$

$$\begin{aligned}
\text{solve_conj}(\perp, _) &= \perp \\
\text{solve_conj}(\text{Sol}, C_1 \wedge \dots \wedge C_n) &= \text{solve_conj}(\text{solve}(\text{Sol}, C_1), C_2 \wedge \dots \wedge C_n) \\
\text{solve_conj}(\text{Sol}, C) &= \text{solve}(\text{Sol}, C)
\end{aligned}$$

4.4. ábra. A korlátozások megoldásának algoritmus

5. fejezet

A success typing Erlanghoz

Mivel az Erlang magnyelv nem tartalmazza az Erlang minden kifejezését, és tartalmaz olyan szabályokat, amelyek nincsenek az eredeti nyelvben (pl `let` és `letrec` kifejezések), ezért a korlátozásgenerálást ki kell egészíteni, illetve átalakítani. Azonban az Erlang tartalmaz több olyan elemet, amelyek nincsenek a funkcionális nyelvekben sem, ezért az adaptálás nem csak a valós Erlang kifejezések és a magnyelv közötti megfeleltetésből áll.

Erlangban minden kifejezés helyén állhat szekvencia kifejezés, ami megnehezíti a típuskörnyezetek használatát. Ugyanis a szekvencia minden utasítása után bővíteni kell a típuskörnyezetet az adott utasításban lévő kötetlen változókkal. Ez nagyon megnehezíti mind az elmélet, mind a szabályok megalkotását. További nehézséget jelent, hogy egy szintaxis vezérelt algoritmus esetében a változó hatókörének és előfordulási helyeinek meghatározása nehéz feladat.

Ahhoz, hogy megfelelően tudjuk kezelni a hasonló problémákat, olyan reprezentációra van szükségünk, amely támogatja szemantikus információk kinyerését. Olyan szemantikus információkra van szükség, amelyek tartalmazzák a változókötéseket és a változók hatókörét. Ezek ismeretében a szabályok kezelése és azok szerkezete lényegesen leegyszerűsödik, ugyanis nem a szintaktikus változó csúcsokhoz kell a típusváltozókat rendelni. A szintaktikus változókhoz hozzárendelünk szemantikus változó objektumokat, amelyek magukban hordozzák a változó előfordulásait, ha-

tókört. Így a különböző hatókörben lévő, de ugyanolyan nevű változókhoz különböző, míg az egy hatókörben lévő azonos nevű változókhoz ugyanaz a típusváltozó fog hozzárendelődni. Ezzel fölöslegessé válik a típuskörnyezet bővítése a szekvencia minden utasítása után, mivel a szemantikus információ már az elemzés kezdetén rendelkezésre áll. Jelölje $\mathcal{S}$ a szemantikus információkkal kiegészített típuskörnyezetet.

A későbbiekben ismertetett (6. fejezet) RefactorErl keretrendszer rendelkezik a szintaxisfa változó csúcsaihoz szemantikus információval, így célszerű lesz az elemzéshez azt használni.

5.1. Korlátozásgenerálás

Az előző szakaszban ismertetett korlátozásgenerálás szabályait megadjuk az Erlang nyelv kifejezéseire. Az egyes kifejezések szintaktikáját a korlátozásgenerálási szabályok mellett adjuk meg.

Egyszerű kifejezések

Az olyan egyszerű kifejezések, mint az egész és valós számok, atomok, karakterláncok, üres listák és tuple¹-ök típusát közvetlenül meg tudjuk határozni. Ezekhez a kifejezésekhez nem generálunk korlátozásokat.

$$\overline{\mathcal{S} \vdash e : \tau_e, \emptyset}$$

Szekvencia kifejezés

A szekvencia

Expr1,
...
ExprN

¹A tuple a rendezett n -est jelenti

több Erlang kifejezésből áll. Az egyes végrehajtási ágak törzse lehet szekvencia kifejezés. A visszatérési értéke megegyezik az utolsó kifejezés visszatérési értékével.

$$\frac{\mathcal{S} \vdash E_1 : \tau_1, C_1; \dots; E_n : \tau_n, C_n}{\mathcal{S} \vdash E_1, \dots, E_n : \tau_n, C_1 \wedge \dots \wedge C_n}$$

A korlátozáshalmaz az egyes kifejezések korlátozáshalmazainak a konjunkciója.

Blokk kifejezés

A blokk kifejezés

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

minden olyan helyen állhat, ahol egyszerű kifejezés. A visszatérési értéke az utolsó kifejezés visszatérési értéke.

$$\frac{\mathcal{S} \vdash E_1 : \tau_1, C_1; \dots; E_n : \tau_n, C_n}{\mathcal{S} \vdash \text{begin } \dots \text{ end} : \tau_n, C_1 \wedge \dots \wedge C_n}$$

A blokk kifejezés szemantikája megegyezik a szekvencia kifejezéssel.

Tuple kifejezés

A tuple-ök

```
{Expr1, ..., ExprN}
```

meghatározott számú kifejezést tartalmaznak. A tuple típusának meghatározáskor meg kell határozni minden egyes elemének a típusát, ezek ismeretében az egész kifejezés típusa az elemek típusából képzett tuple típus. A típusokon nem lehet összevonást végezni. A korlátozásoknak a

konjunkcióját kell venni.

$$\frac{\mathcal{S} \vdash e_1 : \tau_1, C_1; \dots; e_n : \tau_n, C_n}{\mathcal{S} \vdash \{e_1, \dots, e_n\}, \text{tuple}(\tau_1, \dots, \tau_n), C_1 \wedge \dots \wedge C_n}$$

Lista kifejezés

A listák

[Expr1, ... , ExprN]

változó hosszúságú adatszerkezetek. A lista elemei lehetnek inhomogén típusúak, azaz nem feltétlenül kell azonos típusból kikerülniük. Szemben más funkcionális nyelvekkel, mint a Haskell, Erlangban nincsenek végtelen listák. A lista felépülhet úgy, hogy a végéhez egy listát fűzünk.

[Expr1, ... , ExprN | List]

Ilyen esetben a lista konkatenálást feloldjuk, azaz a vég listában lévő elemeket is hozzáillesztjük az eredeti listához, és így határozzuk meg a lista típusát.

$$\frac{\mathcal{S} \vdash e_1 : \tau_1, C_1; \dots; e_n : \tau_n, C_n}{\mathcal{S} \vdash [e_1, \dots, e_n], \text{list}(\tau_1 \cup \dots \cup \tau_n), C_1 \wedge \dots \wedge C_n}$$

A lista típusa egy unió típus, amit a lista elemeinek típusaiból képezünk. Erre azért van szükség, hogy elkerüljük a végtelen adatszerkezet. Ilyen akkor fordulhat elő, amikor a listát egy rekurzív függvénnyel építjük. A korlátozások konjunkciójával a legtöbb információt megőrizzük a kifejezésből.

Listagenerátor kifejezés

A listagenerátor egy olyan lista, ahol az elemeket nem közvetlenül felsoroljuk, hanem valahány másik listából képezzük. Az ilyen listák is végesek, de ezt statikus elemzéssel nehéz kideríteni². A listagenerátor szintaktikája a következő:

²Adatfolyam-elemzéssel néhány esetben lehetséges meghatározni a lista hosszát, de ez koránt sem minden esetben lehetséges.

[Expr || Generator1, ... , GeneratorN , Guard]

Az i -ik generátor (Generator) kifejezés

GeneratorI ::= PatternI <- ListI

egy listából (ListI) és egy mintából (PatternI) áll. Az őrfeltétel (Guard) egy logikai kifejezés, ami a generált elemek közül csak azokat az elemeket hagyja meg, amelyekre a feltétel logikai kifejezés igazat ad. A lista fejében álló kifejezés (Expr) egy tetszőleges Erlang kifejezés lehet.

$$\frac{\mathcal{S} \vdash P_i : \tau_{P_i}, C_{P_i}; L_i : \tau_{L_i}, C_{L_i}; G : \tau_G, C_G; E : \tau_E, C_E}{\mathcal{S} \vdash [|||] : [\tau], \tau_G \subset bool() \wedge (\bigwedge_{i=1}^n (\tau_{L_i} \subset list() \wedge C_{P_i} \wedge C_{L_i})) \wedge C_G \wedge C_E}$$

Az őrfeltételből több is lehet, és elhelyezkedhetnek a generátorok között is. Mivel több őrfeltétel esetén mindegyiknek teljesülnie kell, ezért ezeket konjunkcióval összefogjuk egyetlen feltétellé. Így már alkalmazható lesz rá a levezetési szabály. Miután meghatároztuk a minták és listák típusait és korlátozáshalmazait, meghatározzuk a fejelem típusát és korlátozásait. A listagenerátor típusa a fejelem típusából álló lista lesz. A kifejezésen belüli összes kifejezés korlátozáshalmazának konjunkcióját véve, a lehető leg-pontosabb eredményt kapjuk.

Ennek a kifejezésnek a meghatározása a szemantikus típuskörnyezet nélkül nehéz lett volna. Minden generátor típuskörnyezete öröklí az őt megelőző generátor típuskörnyezetét, és azt még bővíteni kell a generátor mintájában szereplő még kötetlen változókhoz tartozó típusváltozókkal.

Változó kifejezés

A változók esetében egy új típusváltozót adunk vissza, mint visszatérési típus, és nem generálunk megszorításokat. A típusváltozót a szemantikus változó objektumhoz rendeljük hozzá.

$$\overline{\mathcal{S} \vdash v : \tau_v, \emptyset}$$

Mintaillesztő kifejezés

A mintaillesztő kifejezés³

Expr1 = Expr2

akkor és csak akkor lehet sikeres, ha a bal és a jobb oldal típusa azonos. Mivel a nyelvben nincs automatikus típuskonverzió, így nem elég, ha az egyik kifejezés a másiknak altípusa.

$$\frac{\mathcal{S} \vdash e_1 : \tau_1, C_1; e_2 : \tau_2, C_2}{\mathcal{S} \vdash e_1 = e_2 : \tau_2, (\tau_1 \subset \tau_2) \wedge (\tau_2 \subset \tau_1) \wedge C_1 \wedge C_2}$$

Miután meghatároztuk mindkét kifejezés típusát és korlátozást, a mintaillesztő kifejezés típusa a második kifejezés (Expr2) típusa lesz. A két korlátozást konjunkcióját kibővítjük a két típus egyezésének feltételével.

Függvény alkalmazás

Az Erlang nyelvben lévő függvényhívás kifejezés szemantikája és a korlátozásgeneráláshoz használt levezetési szabály megegyezik a magnyelvben ismertett szabállyal. Az egyetlen változtatás, hogy a hagyományos típuskörnyezet helyett a szemantikus típuskörnyezetet használjuk.

$$\frac{\mathcal{S} \vdash f : \tau_f, C_f; a_1 : \tau_1, C_1; \dots a_n : \tau_n, C_n}{\mathcal{S} \vdash f(a_1, \dots, a_n) : \beta, \tau_f = ((\alpha_1, \dots, \alpha_n) \rightarrow \alpha) \wedge (\beta \subset \alpha) \wedge (\tau_1 \subset \alpha_1) \wedge \dots \wedge (\tau_n \subset \alpha_n) \wedge C_1 \wedge \dots \wedge C_n}$$

Az őrfeltételek és a beépített függvények kapcsolata. Az Erlang nyelv rendelkezik néhány előre definiált függvénnyel. Ezek közül néhány egy változó típusának eldöntésére szolgál. Ilyen például az `is_integer/1` függvény. Amennyiben ilyen függvények őrfeltételben fordulnak elő, akkor ezek

³Angolul match expression.

elemzése más szemantikát igényel, ugyanis az ilyen függvényeket tetszőleges típusú paraméterrel meg lehet hívni, és a visszatérési érték típusa minden esetben logikai. Ha ezt a fenti függvényhívás szabállyal típusoznánk, akkor nem nyernénk semmilyen hasznos információt. A függvény nevéből generálunk korlátozásokat a paraméter típusára. Így például az `is_integer(X)` esetén τ_X -el jelölve az X változó típusát a $\tau_X \subset \text{int}()$ korlátozás generálódik.

case kifejezés

A case kifejezés

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

nagyon hasonlít az Erlang magnyelvben megtalálható case kifejezéshez, de ez esetben az őrfeltételnek nem felétlenül kell jelen lennie. Abban az esetben, amikor nincs jelen az őrfeltétel, akkor a magnyelvben a 'true' feltételt helyettesítjük.

$$\frac{\mathcal{S} \vdash e : \tau_e : C_e; p_1 : \tau_{p_1}, C_{p_1}; \dots; p_n : \tau_{p_n}, C_{p_n}; b_1 : \tau_{b_1}, C_{b_1}; \dots; b_n : \tau_{b_n}, C_{b_n}}{\mathcal{S} \vdash \text{case } e \text{ of } p_1 \rightarrow b_1; \dots; p_n \rightarrow b_n \text{ end} : \bigcup_{i=1}^n \tau_{b_i}, (\tau_e \subset \bigcup_{i=1}^n \tau_{p_i}) \wedge (\bigvee_{i=1}^n C_e \wedge C_{p_i} \wedge C_{b_i})}$$

A korlátozás generálási szabályban a típuskörnyezetre vonatkozó részt módosítottuk csak. Amennyiben hiányzik valamelyik őrfeltétel, akkor értelem szerűen azt a részt kihagyjuk a szabályból.

Infix és prefix kifejezések

A függvényhívás egy speciális esete az infix operátort tartalmazó kifejezés. A korlátozás generálási szabály ennek megfelelően egy aktualizált változat, melyben az operandusok tényleges és elvárt típusának ismeretében behelyettesítettünk az függvény alkalmazás szabályába. Jelöljön $\star$ egy tetszőleges kétoperandusú infix műveletet.

$$\frac{S \vdash e_1 : \tau_{e1}, C_1; e_2 : \tau_{e2}, C_2}{S \vdash e_1 \star e_2 : \tau, (\tau_{e1} \subset \tau_1) \wedge (\tau_{e2} \subset \tau_2) \wedge C_1 \wedge C_2}$$

Az Erlangban a programozó nem definiálhatja felül operátorokat, továbbá új infix műveleteket sem vezethet be. Ezért az algoritmus ismeri az összes Erlangban megtalálható operátor típusspecifikációját.

A prefix operátorokra elmondható ugyanez. Jelöljön $\star$ egy tetszőleges unáris operátort.

$$\frac{S \vdash e_1 : \tau_{e1}, C_1}{S \vdash \star e_2 : \tau, (\tau_{e1} \subset \tau_1) \wedge C_1}$$

Üzenet küldő kifejezés. Az Erlang nyelv erőssége a párhuzamos végrehajtás. A folyamatok egymással üzenetekkel kommunikálhatnak. Ehhez egy infix operátort biztosít a nyelv:

`PidSpec ! Expr`

A `PidSpec` egy folyamatot azonosít. Ennek a folyamatnak tetszőleges termet küldhetünk. Az üzenetküldés csak abban az esetben nem teljesül, ha a folyamatot egy nem bejegyzett névvel próbáljuk meg elérni. Minden más esetben az üzenetküldés sikeres, és a visszatérési értéke az elküldött üzenet. Azt, hogy a program egy adott pontján a folyamat neve be van-e jegyezve statikus elemzéssel nem lehet kideríteni, így erre az esetre nem generálunk korlátozást.

$$\frac{S \vdash P : \tau_P, C_P; E : \tau_E, C_E}{S \vdash P!E : \tau_E, C_P \wedge C_E \wedge (\tau_P \subset \text{pidspec}())}$$

Függvény kifejezés

Az Erlang magnyelvvel ellentétben az Erlang függvény kifejezésnek lehet több ága. A függvényhez hozzátartozik a paramétereinek száma. A paraméterek tartalmazhatnak mintaillesztést is.

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

Amennyiben rendelkezünk a paraméterek, az őrfeltételek és a függvénytörzsek típusaival és azok korlátozáshalmazaival, akkor a függvény kifejezés típusa egy olyan nyíl típus, ahol az i -ik paraméter típusa a függvényágak i -ik paraméterének típusaiból képzett unió típus, míg a visszatérési érték típusa az ágak visszatérési típusaiból képzett unió. Az egyes függvényágak korlátozáshalmazainak konjunkcióját kell venni, majd ezeknek a diszjunkcióit. A függvény kifejezés szabálya hasonlít a case kifejezés szabályára.

$$\frac{\mathcal{S} \vdash P_{ij} : \tau_{ij}, C_{ij}; G_j : \text{bool}(), C_{B_j}; B_j : \tau_j, C_j}{\mathcal{S} \vdash \text{fun} : \left(\bigcup_{j=1}^m \tau_{j1}, \dots, \bigcup_{j=1}^m \tau_{jn} \right) \rightarrow \bigcup_{j=1}^m \tau_j, \bigvee_{j=1}^m \left(\bigwedge_{i=1}^n C_{ij} \wedge C_{G_j} \wedge C_j \right)}$$

if kifejezés

Az if kifejezés

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

BodyN
end

több ágból áll, mindegyik ághoz tartozik egy őrfeltétel, aminek a teljesülése esetén az aktuális ág törzse végrehajtható.

$$\frac{\mathcal{S} \vdash G_1 : \tau_{G_1}, C_{G_1}; \dots; G_n : \tau_{G_n}, C_{G_n}; B_1 : \tau_{B_1}, C_{B_1}; \dots; B_n : \tau_{B_n}, C_{B_n}}{\mathcal{S} \vdash \text{if} : \bigcup_{i=1}^n \tau_{B_i}, \bigwedge_{i=1}^n (\tau_{G_i} \subset \text{bool}()) \wedge (\bigvee_{i=1}^n (C_{G_i} \wedge C_{B_i}))}$$

Miután meghatároztuk az őrfeltételek és a hozzájuk tartozó törzsek típusát és a korlátozás halmazokat, az if kifejezés típusa a törzsek típusainak uniója. Minden őrfeltétel típusa valamilyen logikai érték kell legyen. Az ebből képzett korlátozás halmazhoz hozzáadjuk az egyes feltételek és a hozzájuk tartozó törzsek korlátozásainak konjunkcióinak diszjunkcióját.

Egyéb Erlang kifejezések

A többi Erlang kifejezés visszavezethető az eddig ismertett kifejezéstípusokra. A receive és try kifejezés a case kifejezésre.

5.2. Egységesítés

Az egységesítő algoritmus megegyezik az Erlang magnyelvben bemutatott algoritmussal, ugyanis sem a típusrendszeren, sem pedig a korlátozáshalmazok szerkezetén nem változtattunk. Ez azt is jelenti, hogy annak az algoritmusnak helyessége továbbra is megmarad.

5.3. Példa típusozás

A következőkben bemutatom a fenti algoritmust egy egyszerű és egy bonyolultabb példán. Ez első példán a korlátozásgenerálást fogom részletezni, majd a második példán az egyesítő algoritmus működését szemléltetem.

5.3.1. Korlátozásgenerálás

Tekintsük a következő egyszerű függvényt:

`f(X) when is_integer(X) -> X+2.`

Ahhoz, hogy meghatározzuk a függvényhez tartozó korlátozáshalmazt ismerünk kell a paraméter, az őrfeltétel és a függvénytörzs típusát és a hozzájuk tartalmazó korlátozáshalmazt. Mivel X ugyanazt a változót jelenti, ezért a hozzá tartozó szemantikus változó objektumból származó típusváltozó legyen τ_X .

A függvénytörzs levezetéséhez először alkalmazzuk a változó, majd az egyszerű kifejezés levezetési szabályát. Ezek után az infix kifejezés szabályába beírjuk a kapott értékeket.

$$(*) := \frac{\overline{\mathcal{S} \vdash X : \tau_X, \emptyset} \quad \overline{\mathcal{S} \vdash 2 : int(), \emptyset}}{\mathcal{S} \vdash X+2 : number(), \emptyset}$$

Az őrfeltétel levezetéséhez elegendő a változó szabályt alkalmazni, majd az őrfeltételekhez alakított függvényhívás szabályát alkalmazzuk.

$$(**) := \frac{\overline{\mathcal{S} \vdash X : \tau_X, \emptyset}}{\mathcal{S} \vdash is_integer : bool(), (\tau_X \subset int())}$$

A függvény kifejezés levezetéséhez még egyszer alkalmazzuk a változó szabályt és felhasználjuk az előző eredményeket.

$$\frac{\overline{\mathcal{S} \vdash X : \tau_X, \emptyset} \quad (*) \quad (**)}{\mathcal{S} \vdash f : (\tau_X) \rightarrow number(), (\tau_X \subset int())}$$

Megoldva a korlátozáshalmazt az f függvényre a $(int()) \rightarrow number()$ típus adódik.

5.3.2. Második példa

A következő szakaszban bemutatom egy egyszerű függvény típusspecifikációjának kiszámítását. Tekintsük a következő függvényt:

```

drop_odd([]) ->
  [];
drop_odd([_]) ->
  [];
drop_odd([_, H | T]) ->
  [ H | drop_odd(T) ].

```

A case kifejezés első és második ágához tartozó korlátozáshalmazok triviálisak. A lényeg a harmadik klózban található függvényhívás típusozása. Jelölje τ_D a `drop_odd` függvényhez tartozó rekurzív típusváltozót. A szabályok alkalmazásával a

$$\overline{A \vdash \text{drop_odd}(T) : \beta, C_1 = ((\tau_D = (\alpha_1) \rightarrow \alpha) \wedge (\beta \subset \alpha) \wedge (\tau_T \subset \alpha_1))}$$

következtetéshez jutunk. Mivel a függvény rekurzív, ezért helyezzük a megoldáshalmazba a $\tau_D \mapsto \text{none}()$ megfeleltetést. Ezzel garantáljuk, hogy az egyesítő algoritmus legalább egyszer iterálni fog. Jelölje a case kifejezés i -ik diszjunkciójához tartozó megoldáshalmazt Sol_i , a teljes függvényhez tartozó megoldáshalmazt Sol , a case kifejezés visszatérési típusát τ_c . Ekkor az első iterációban a következő megoldáshalmazokat kapjuk.

$$\begin{aligned}
Sol_1 &= \{\tau_L \mapsto [], \tau_c \mapsto []\} \\
Sol_2 &= \{\tau_L \mapsto \text{list}(), \tau_c \mapsto []\} \\
Sol_3 &= \perp \\
Sol &= Sol_1 \sqcup Sol_2 \sqcup Sol_3 = \{\tau_L \mapsto \text{list}(), \tau_c \mapsto []\}
\end{aligned}$$

A rekurzív típusváltozó a

$$\tau_D = (\tau_L) \rightarrow \tau_c = (\text{list}()) \rightarrow []$$

értéket veszi fel, ami változást jelent, így újra végrehajtjuk az algoritmust. A típuskörnyezet jelenleg a következő képen alakul:

$$\{\tau_D \mapsto \tau_c = (list()) \rightarrow [], \alpha_1 \mapsto list(), \alpha \mapsto [], \beta \mapsto [], \tau_T \mapsto list()\}.$$

A korlátozáshalmaz nem változott. Így a diszjunkciók megoldásai a következő képen alakulnak:

$$\begin{aligned} Sol_1 &= \{\tau_L \mapsto [], \tau_c \mapsto []\} \\ Sol_2 &= \{\tau_L \mapsto list(), \tau_c \mapsto []\} \\ Sol_3 &= \{\tau_L \mapsto list(), \tau_c \mapsto list()\} \\ Sol &= Sol_1 \sqcup Sol_2 \sqcup Sol_3 = \{\tau_L \mapsto list(), \tau_c \mapsto list()\} \end{aligned}$$

Ezzel a rekurzív típusváltozóhoz a

$$\tau_D = (\tau_L) \rightarrow \tau_c = (list()) \rightarrow list()$$

típust rendeltük. Ez az érték különböző az előzőtől, azonban az algoritmus újbóli futása nem hoz új eredményt, tehát a függvényt sikerült típusoznunk.

6. fejezet

Típuskikövetkeztető rendszer a RefactorErlben

A RefactorErl egy az Eötvös Loránd Tudományegyetemen fejlesztett statikus forráskód elemző és újrastrukturáló eszköz Erlang nyelvhez. A kód elemzése során szintaxisfát épít, és erre épülve egy szemantikus gráfot. Ezek között megtalálható függvényhívási gráf, a változók hatókörét és kötését tartalmazó gráf, adatfolyam gráf. Mivel a rendszerbe betöltött fájlok esetében már rendelkezünk a fenti információkkal, ezért célszerű a success typing algoritmusát adaptálni a RefactorErlhez.

A RefactorErl keretrendszerhez készítettem egy típusspecifikáció generáló komponenst. Ez a komponens az eddig ismertett success typing algoritmust használja fel, hogy Erlang nyelven írt függvények specifikációját meghatározza. Ebben a fejezetben ismertetem a megvalósítás során felmerült problémákat és az azokra adott megoldást.

6.1. Első prototípus

Az első prototípusban még nem használtam a korlátozás generálás, egyesítés módszerét, hanem a RefactorErl szintaktikus gráfját jártam be, és az alapján következtettem a kifejezés típusára. Egy kifejezés esetén a kikövetkeztetett típussal szűkítettem a paraméterek típusát.

Ezzel a módszerrel több komoly probléma is volt. Az első problémát a case kifejezés típusozása jelentette. Egy klózon belül nem lehetett szűkíteni egy paraméter típusát, ugyanis nem lehetett tudni, hogy az unió típusnak melyek azon részei, amelyet nem érint az adott klózban végzett korlátozás. A legnagyobb problémát az inhomogén visszatérési érték és a rekurzió jelentette. Mivel nem készítettem korlátozáshalmazokat, ezért rekurzió esetén többször kellett elemezni a forráskódot. Ez az inhomogén visszatérési értékkel együtt nagyon bonyolult esetek vizsgálatát követelte volna meg.

Ezt a módszert elvetettem, és megvizsgáltam más típuskikövetkeztési módszereket is.

6.2. A success typing-on alapuló prototípus

A típuskikövetkeztető rendszert újraimplementáltam a success typing módszerrel követve.

Az elemzés kiindulópontja egy forrásnyelvi függvénydefiníció. A specifikáció generálása három fázisból áll: először a korlátozáshalmazokat állítjuk elő, majd a kapott korlátozáshalmazból az egyesítő algoritmussal meghatározzuk a függvény specifikációját, majd a specifikáció ábrázolásához használt belső adatszerkezetet a felhasználó igényei szerint megjelenítjük.

Rekurzív függvények kezelése. A korlátozásgenerálás első fázisában meg kell határozni, hogy az adott függvény része-e egy rekurzív függvényhívási gráfnak. Amennyiben igen, akkor a láncban szereplő függvényeknek egy ideiglenes típust $((none(), \dots, none()) \rightarrow none())$ határozzunk meg. Erre azért van szükség, mert az eredeti függvény törzsében lévő függvényhívások elemzésnél szükséges információ, hogy az adott függvény típusát meg kell-e még határozni. Ha a specifikációval még nem rendelkezünk, akkor arra a függvényre is lefuttatjuk az algoritmust. Rekurzív függvény esetében ez egy soha nem termináló folyamat lenne, de az ideig-

lenes típussal elkerülhetjük a végtelen rekurziót. A RefactorErl rendelkezik egy olyan komponenssel, ami képes meghatározni a függvényhívási gráfnak a ciklikus részeit. Ezt felhasználva a fenti probléma egyszerűen kezelhető. A több klózból álló függvényekre klózonként külön futtatjuk a korlátozásgenerálást, majd az így keletkezett halmazokat összevonjuk.

Paraméterek kezelése. Először a függvény paramétereinek a korlátozásait állítjuk elő. A paramétereket egymástól függetlenül elemezzük, majd a végén a korlátozáshalmazokat összevonjuk. A paraméterek visszatérési értékét eltároljuk.

Őrfeltételek kezelése. Az őrfeltételek elemzését külön implementáltam, mivel az őrfeltételben szereplő függvényhívások szemantikája eltér a szokásos függvényhívásokétól. A paraméterek elemzése után kapott korlátozáshalmazt bővítem az őrfeltétel korlátozáshalmazával.

A függvénytörzs kezelése. Az őrfeltételek elemzése után keletkezett korlátozáshalmazt tovább bővítjük a függvény törzséből származó korlátozásokkal. A függvénytörzs utolsó kifejezésének típusa lesz a függvényág visszatérési típusa.

Egyesítés. A megoldáshalmaz a függvény kifejezéseinek és a hozzájuk kapcsolódó típusoknak a rendezett halmaza. A megoldáshalmazhoz hozzáadjuk a rekurzív hívási láncban lévő függvények típusváltozóit, majd ezzel és az előbb generált korlátozáshalmazzal lefutattjuk a 4.4. ábrán ismertetett egyesítő algoritmust. Az egyesítő algoritmus felhasználja a *Sol* leképezést, ami a típusváltozók aktuális konkrét típusát adja vissza. Az algoritmus végén a megoldáshalmazból ki kell olvasni a típusozni kívánt függvény specifikációját.

Eredmény megjelenítése. A specifikáció belső reprezentációja a programozó számára nehezen olvasható, ezért a kész specifikációt transzformálni kell a felhasználó számára olvasható formátumba. Ez történhet szö-

veges (sztring) formába, ami megfelel a függvnyspecifikációk számára ajánlott szerkezettel. Lehetőség van továbbá a specifikáció forrásprogramba történő beszúrására is.

Implementációs megjegyzések

Az Erlang nyelvben nem lehet operátorokat felüldefiniálni és nem lehet új infix vagy prefix operátorokat létrehozni. Ezért a megoldó programban az operátorok specifikációja előre megadott. Ugyanígy jártunk el a beépített függvények esetében is, ugyanis ezek nem Erlang nyelven vannak megvalósítva, így még lehetőségünk sincs az algoritmussal típusozni őket. Ugyanakkor az alkalmazáskönyvtár (OTP) interfész függvényeihez adottak a specifikációk, amiket feldolgozva lehetőség van növelni a típusozás pontosságán.

Amikor olyan függvényhíváshoz érünk, amihez nem rendelkezünk a függvény forráskódjával, azt feltételezzük, hogy a függvényt tetszőleges típusú (*any()*) paraméterekkel meg lehet hívni, és a visszatérési érték is tetszőleges típusú lehet. Egy függvény definíciója három okból hiányozhat:

1. A felhasználó nem töltötte be a kódot a RefactorErl rendszerbe. Például, mert az adott modul még nincs megírva, vagy mert a program egyes részei nem lényegesek. Ezt célszerű elkerülni a minél pontosabb eredmény érdekében.
2. Az Erlang forráskód nem elérhető mivel egy harmadik fél készítette, és mi már csak a lefordított fájlokat kaptuk meg. A RefactorErl-hez készült egy komponens, ami lehetővé teszi az ilyen fájlok betöltését.
3. Az adott funkció nem Erlang nyelven történt megírásra. Ilyen esetben nincs lehetőség az elemzést lefuttatni rajtuk.

A programozó megadhatja kézzel a függvények specifikációját függetlenül attól, hogy tartozik-e hozzájuk definíció vagy sem. Ez esetben az algoritmus a felhasználó által megadott specifikációt tekinti helyesnek.

6.3. Eredmények a RefactorErlben

Ebben a szakaszban ismertetem a 6. fejezetben bemutatott RefactorErlben elkészített success typing-on alapuló implementációval kapcsolatos eredményeket.

Az elkészült program képes Erlang nyelven írt függvények specifikációjának meghatározására, az eredmények szöveges megjelenítésére vagy a forráskódba ágyazására.

Tekintsük a következő példát:

```
f(X,Y) when is_integer(X) ->
    X=Y+2,
    Y=Y / 3;
f(X,Y) ->
    Y=alma,
    g(X),
    nok.

g([]) ->
    korte.
```

A függvényhez generált specifikáció a következő:

```
f/2 :: ((integer()|nil()), (atom(alma)|float()))
      -> (atom(nok)|float())
```

Bár a fenti függvény nem csinál sok hasznosat több kifejezés is megtalálható benne. Az első klózban az $Y+2$ és az $Y/3$ kifejezésekben az Y egész és lebegőpontos (float) értéket is felvehet. Azonban a második mintaillesztés kifejezésben az osztás visszatérési típusával kell megegyezzen. Ez pedig csak lebegőpontos lehet. Az X változóra vonatkozó egész értékű korlátozás leszűkíti az összeadás visszatérési típusát, ami teszőleges egész lehet. A klóz visszatérési értéke az $Y/3$ kifejezés értéke, ami lebegőpontos szám típusú.

A második klózban az X változótípusára csak a $g/1$ függvényhívás kifejezésből tudunk következtetni. Ahhoz, hogy a függvényhívás ne végződjön futás idejű hibával, az X csak az üres listát ($nil()$) veheti fel. Az $Y=alma$ mintaillesztés csak akkor lehet sikeres, ha az Y az $alma$ atom értéket veszi fel. Ennek a klóznak a visszatérési értéke a nok atom.

A következő példában egy rekurzív függvényt láthatunk:

```
drop_odd([]) ->
  [] ;
drop_odd([_]) ->
  [] ;
drop_odd([_, H | T]) ->
  [H | drop_odd(T)] .
```

A függvény a paraméterül kapott listából elhagyja a páratlan helyen álló elemeket. A kiszámított specifikáció a következő:

```
drop_odd/1 :: ([any()]) -> [any()]
```

6.3.1. Összehasonlítás a TypEr-rel

Heurisztika

A TypEr jelenlegi változata a succes typing egy heurisztikával megerősített változatát használja. Ez a heurisztika abból áll, hogy a nem exportált függvények specifikációját pontosítja úgy, hogy felderíti a függvény alkalmazásait, majd az aktuális paraméterekből megpróbálja kitalálni a programozó szándékát. Azaz, ha egy függvényt a succes typing szerint tetszőleges típusú paraméterrel meg lehet hívni, és az nem exportált, illetve például csak listával hívjuk meg, akkor feltételezi, hogy a függvényt úgy tervezték, hogy listákon működjön. Ekkor a specifikációt módosítja eszerint az információ szerint.

Ezzel a heurisztikával sokat lehet javítani az algoritmus pontosságán, ugyanakkor az a mellékhatása, hogy a generált specifikáció félrevezetheti a programozót a függvény használatáról. A RefactorErl-ben megvalósított

megoldásnak nem célja ilyen heurisztika alkalmazása. Ez természetesen azt is jelenti, hogy az így generált specifikáció sokkal kevésbé lesz pontos, ezáltal a megtalált típushibák száma is kisebb.

A TypEr rendelkezik egy olyan heurisztikával is, amely felfedi az elérhetetlen kódrészeket. Ezen részeknek nem lesz hatásuk az elemzésre, így az elemzés végeredménye pontosabban tükrözi az adott függvény működését. Ilyen elemzés még nincs megvalósítva a RefactorErl keretrendszerben, így az ezen alapuló típuskikövetéztető algoritmus sincs erre felkészítve.

Típushibák

A 4.4. ábrán ismertetett algoritmus hivatott kimutatni, ha típushiba történik. Ez azonban csak annyit tud megmutatni, hogy egy függvényben valahol típushiba történik, ugyanis hiba esetén az üres megoldáshalmazzal ($\perp$) tér vissza. Mivel a hiba helyének megtalálásához több információt kellene az algoritmusnak átadni vagy az algoritmuson úgy változtatni, hogy hiba esetén csak jelölje a hibás kifejezéseket, majd ezzel számoljon tovább. Ez a módosítás azonban nem triviális, ugyanis az algoritmus terminálási feltétele, hogy a megoldáshalmaz ne módosuljon egy újbóli futtatás során. Az új futás során azonban a típushibás kifejezés típusa hatással lesz a többi kifejezés típusára is, ami szintén azt eredményezi, hogy azok a kifejezések is hibásnak fognak mutatkozni.

6.4. Fejlesztési lehetőségek

Az elemzés memóriafelhasználásán azáltal lehet javítani, hogy az elemzés elején felépítjük az adott függvényből kiinduló függvényhívási gráfot, meghatározzuk a rekurzív hívási köröket, majd ezeket egy csúcsba vonjuk össze. Ezután az algoritmust az így keletkezett körmentes irányított gráf (DAG) leveleitől kezdjük elemezni. Ezáltal egy függvény törzsében lévő függvényhíváskor már rendelkezni fogunk az adott függvény specifikációjával.

További fejlesztési lehetőség a hibák helyének meghatározása. Ehhez módosítani kell az egyesítő algoritmuson.

Az algoritmus pontosságának javítására irányul az a kutatás, amely a success typing módszert kibővítené úgy, hogy az elemzés alapegysége ne egy függvény legyen, hanem egy klóz. Ezzel egy lépést tennénk a feltételes típusok felé, azonban meg szándékozunk őrizni az olvashatóságot a pontosság növelése mellett.

A RefactorErl keretrendszer alkalmas a forráskód újrastrukturálásra (refaktorálására). Egy újabb fejlesztési lehetőség a refaktorálás során végrehajtott módosítások alkalmazása a specifikációkra a függvények újraelemzésének szükségessége nélkül. Vagy szűkítve azon függvények körét, melyek újraelemzést igényelnek.

7. fejezet

Összefoglalás

Dolgozatomban bemutattam a típuskikövetkeztetés elméletét. Egy konkrét programozási nyelv, az Erlang esetén ismertettem egy kifejezetten erre a programnyelvre alkotott típuskikövetkeztetési módszert. Ez a módszer a success typing. Ennek a módszernek nagy hátránya, hogy nem az Erlang forrásállományokon végzi az elemzést, hanem egy olyan köztes tisztán funkcionális programnyelven, amit az Erlang fordítója készít, az Erlang magnyelven. Ismertettem egy az ipari környezetben is használt eszközt, amely ezen a köztes nyelven típuskikövetkeztetést végez. Ez az eszköz a TypEr.

A dolgozatomban ismertettem a success typing algoritmusnak a kiterjesztését a teljes Erlang nyelvre. Ennek keretén belül megadtam a levezetési szabályokat az Erlang nyelv minden kifejezésére oly módon, hogy általánosítottam a success typing algoritmus típuskörnyezet fogalmán. Az így megalkotott szemantikus információkkal bővített típuskörnyezet egyszerűbbé tette a szabályok megalkotását a használhatóság korlátozása nélkül.

Bemutattam egy prototípus implementációt, melyet a RefactorErl keretrendszer segítségével készítettem el az általam kiterjesztett success typing algoritmus felhasználásával. Az így elkészült eszköz nem igényli a forrásprogram lefordítását, anélkül is helyes eredményt ad. Ez az eszköz messze nem tökéletes, és jelen állapotában nem alkalmas a TypEr helyet-

tesítésére, ugyanakkor bemutattam több olyan fejlesztési lehetőséget, amivel pontosabb eredményt adhat a TypEr-nél.

Irodalomjegyzék

- [1] *Clean homepage*, Nov 2011, <http://wiki.clean.cs.ru.nl/Clean>.
- [2] *Haskell homepage*, Nov 2011, <http://haskell.org/haskellwiki/Haskell>.
- [3] *RefactorErl homepage*, May 2011, <http://plc.inf.elte.hu/erlang/>.
- [4] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman, *Soft typing with conditional types*, Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '94, ACM, 1994, pp. 163–173.
- [5] Joe Armstrong, *Programming Erlang*, Pragmatic Bookshelf, 2007.
- [6] Tobias Lindahl and Konstantinos Sagonas, *Abstract practical type inference based on success typings*, 2005.
- [7] Tobias Lindahl and Konstantinos Sagonas, *Typer: a type annotator of erlang code*, Proceedings of the 2005 ACM SIGPLAN workshop on Erlang (New York, NY, USA), ERLANG '05, ACM, 2005, pp. 17–25.
- [8] Martin Logan, Eric Merritt, and Richard Carlsson, *Erlang and OTP in Action*, Manning, November 2010.
- [9] Simon Marlow and Philip Wadler, *A practical subtyping system for erlang*, SIGPLAN Not. **32** (1997), 136–149.
- [10] Csörnyei Zoltán, *Lambda-kalkulus a funkcionális programozás alapjai*, Typotex, 2007.