Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok

Tanszék

Táblázatkezelő szoftver implementálása Haskell nyelven

Supervisor:
Dr. Kaposi Ambrus
egyetemi docens

Author:
Széles Márk
programtervező informatikus BSc

Budapest, 2021

This page should be the original Thesis Topic Declaration.

Tartalomjegyzék

fejezet 1

Bevezetés

Ha indokolni szeretnénk egy új táblázatkezelő szoftver elkészítésének létjogosultságát, két kérdésre kell választ adnunk:

1. Milyen funkciókat kell ellátnia egy táblázatkezelő szoftvernek?
2. Mi az, ami hiányzik a jelenleg elterjedt szoftverekből? (Pl. Microsoft Excel)

Az első kérdésre talán az a legegyszerűbb válasz, hogy egy táblázatkezelő lehetőséget ad adatok tárolására és a bevitt adataink alapján újabb adatok kiszámítására. Ez a valóságban számtalan alkalmazási lehetőséget jelent. Az Excel-lel például lehet színes, táblázatos formájú órarendet készíteni, egy gyakorlati csoport eredményeit számontartani, családi költségvetést vezetni, stb.

Egy táblázatkezelőben minden cella tartalma egy funkcionális program. Egy cellába írhatunk egy egyszerű kifejezést (adat), vagy egy összetettebb programot, ami korábbi adatok függvényében számít ki egy új adatot. A táblázatkezelő tehát nem más, mint egy könnyen használható interfész a háttérben meghúzódó funkcionális nyelvhez. Ennek a nyelvnek az intuitív használatát számos funkció segíti. Lehetővé válik az összetett program komponensekre bontása, és az egyes komponensek eredményeinek hatékony vizualiuációja.

Ha tekintjük napjaink legnépszerűbb táblázatkezelő szoftverét, az Excel-t, azt láthatjuk, hogy a fent leírt feladatot kiválóan ellátja. Bővelkedik megjelenítéssel kapcsolatos opciókban, a felhasználói felület használata intuitív, az elérhető dokumentáció közérthető. Fő hiányosságát nem is ebben látom, hanem az általa használt programozási nyelvben. Az Excel-ben a szoftver saját programozási nyelvét használhatjuk, aminek bővítésére a VBA programnyelv használatával van lehetőség. *(refe-*

rencia?) Ez azonban nem a legkényelmesebb megoldás, nehézkes egy összetettebb számítási funkciót hozzáadni az eszköztárunkhoz.

Ezen probléma megoldására teszek kísérletet dolgozatomban. Egy olyan táblázatkezelő szoftvert készítettem el, aminek a celláiba – a táblázatkezelő funkciók megfelelő ellátása érdekében kissé kiegészített – Haskell nyelven lehet programokat írni. Így a felhasználó rendelkezéysére áll egy általános célú programnyelv teljes eszköztára.

fejezet 2

Felhasználói dokumentáció

fejezet 3

Fejlesztői dokumentáció

3.1 A fejlesztői dokumentáció felépítése

A fejlesztői dokumentáció három nagy részből áll. Az 3.2 részben ismertetésre kerülnek a szoftver készítése során felhasznált technológiák, valamint nagy vonalakban a szoftver logikai felépítése. (Milyen programkomponensek vannak, milyen feladatokat látnak el, hogyan kapcsolódnak egymáshoz.) A 3.3 rész tartalmazza az egyes komponensek részletesebb leírását. Minden komponens esetén ismertetésre kerül a más komponensek felé nyilvánosságra hozott interfész, valamint a komponensek működési elve, beleértve a használt típusok leírását és a fontosabb algoritmusok működési elvét. A 3.4 rész tartalmazza a tesztelési eljárás leírását és a tesztelés eredményeit.

3.2 A szoftver felépítése

3.2.1 Felhasznált technológiák összefoglalása

Az alkalmazás Haskell nyelven íródott. A grafikus megjelenítés *GTK+* alapú, a *gtk2hs* csomag által biztosított bindingokat használtam a grafikus felület kezeléséhez. Ez a csomag a *GTK+* osztályhierarchiáját Haskell típusosztályok hierarchiájaként adja vissza. Az egyes osztályok metódusainak a típusosztályok definíciójában szereplő függvények felelnek meg. A *GTK+* típusai foreign pointerek segítségével vannak megvalósítva, és *IO*-ban használhatók.

Az alkalmazás a *GTK+* logikájának megfelelően eseményvezérelt. A felhasználó akciói eseményeket váltanak ki, amelyek hatására handlerek futnak le. A handlerek minden esetben *IO* akciók, amelyek valamilyen módon módosítják a globális állapotot (lásd 3.2.1. Globális állapot). A szoftver fejlesztése során fontos volt, hogy minél kevesebb legyen az tisztátalan (impure), *IO*-n belül elvégzett számítás. Igyekeztem a program logikájának minél nagyobb részét egy tiszta, nem monadikus környezetben megvalósítani. Így a számítások helyessége könnyebben tesztelhető/verifikálható, a handlerek már keveset számolnak az *IO*-ban.

Az alkalmazás a parseolási feladatokhoz a *parsec* csomagot, a gráfok kezeléséhez az *fgl* csomagot, a ghci futtatásához pedig a *ghcid* csomagot használja. A modell adatainak könnyebb kezeléséhez a *microlens-platform* csomagot használtam, ami a jól ismert *lens* csomag egy kevesebb funkciót és kevesebb függőséget tartalmazó változata. A függőségek pontos listája elérhető a Felhasználói dokumentációban, illetve az egyes programkomponensek részletes leírásakor is említésre kerülnek a fontosabb felhasznált csomagok.

3.2.2 A globális állapot

Az alkalmazás fő felépítését egy az FP Complete blogján megjelent cikk **(IDE KÉNE EGY REFERENCIA)** inspirálta. Az alkalmazás a globális (olvasható) állapotot a *ReaderT* monád transzformer segítségével valósítja meg, az alkalmazás vezérlése így egy *ReaderT Env IO* környezetben történik, ahol *Env* a globális állapotot leíró adattípus. Fontos megjegyezni, hogy bár az *Env* típus komponensei az inicializálás után sosem módosulhatnak, a mögöttes állapot még változhat, hiszen a komponensek módosítható referenciák. Ez nagyon hasonló a Java nyelvben használható konstans referencia koncepciójához: a referencia nem változhat, de a referált adat igen.

A fentebb referált cikk által inspirálva a (GUI komponensein kívüli) globális állapot egy *StateT* transzformer helyett módosítható referenciákkal (*IORef* és *MVar*) kezeltetik. Ugyanis hiába tiszta, ha globálisan használjuk a *StateT*-t, valójában – a programlogika szintjén – ugyanúgy egy globális, módosítható állapotot vezetünk be. Szintén egy szempont, hogy a *GTK+* alapú *GUI* miatt eleve szerepelnek módosítható referenciák (foreign pointer) a globális állapotban, így ez a probléma sem-

miképpen nem kerülhető el teljesen. Egy további érv a globális *StateT* ellen, hogy egy nagyobb monad stack szükségszerűen bonyolítja a programot. A *ReaderT IO* ellenben még kifejezetten könnyen kezelhető. A cikk konkurrenciához köthető problémákat is említ a *StateT*-vel kapcsolatban. Ez a szoftver jelenlegi verziójában még nem olyan jelentős (lévén a mostani implementáció nagyon kis mértékben épít konkurrenciára). Azonban a jövőre nézve mindenképpen előnyös, ha a szoftvert könnyen lehet a konkurrens paradigma szerint bővíteni.

Ezen bevezető után tekintsük a globális állapot definícióját! Az alábbi típusdefiníció az *App.Types* modulban található:

```
1  data EvalConfig = EvalConfig { modules :: [String]
2                               , paths :: [String]
3                               }
4    deriving (Eq, Generic)
5
6  data EvalControl = EvalControl { eGhci    :: MVar Ghci
7                                 , eCommand :: MVar String
8                                 , eResult  :: MVar (Either String [
                                     String])
9                                 , eConfig  :: MVar EvalConfig
10                                } deriving Eq
11
12 data SaveStatus = Saved | Modified
13   deriving Eq
14
15 data File = File FilePath SaveStatus
16   deriving Eq
17
18 data Env = Env { evalControl  :: EvalControl
19                , gui          :: Gui
20                , state        :: IORef Spreadsheet
21                , file         :: IORef (Maybe File)
22                } deriving Eq
```

Code 3.1: Az Env típus

Az *evalControl* mező tartalmazza a kifejezések ghci-ban való kiértékelés hez szükséges erőforrásokat. Az *eConfig* mező tartalmazza a GHCi-hoz tartozó konfigurációs beállításokat. (Betöltött modulok listája, és a modulok keresési útvonalainak listája.)

Az *eGhci* mező tartalmazza a háttérben futó GHCi példányra való hivatkozást. Az *eCommand* és az *eResult* valósítják meg a kommunikációt a kiértékelést végző szál és az alkalmazás fő szála között. Az *eCommand*-nak a fő szál a termelője, és a kiértékelő szál a fogyasztója, az *eResult*-nak pedig fordítva.

A *gui* mező tartalmazza a GUI komponenseit. A pontos típusdefiníció a GUI leírásánál fog szerepelni.

A *state* mező egy módosítható referencia, ami a számolótáblát reprezentáló, *Spreadsheet* típusú adatot referálja. A file adattag tartalmazza az éppen a táblázatkezelőbe betöltött fájl fontosabb adatait, amennyiben be van töltve egy fájl. (Fájl neve, és állapota.)

Az *EvalConfig* típus példánya *Generic* és *Serialize* típusosztályoknak. Erre a perzisztenciához van szükség.

3.2.3 Programkomponensek és modulszerkezet (nem teljes!!!!!!!!!)

Az alábbiakban röviden összefoglalom a szoftver moduljainak fő feladatát:

- **Main** – főprogram
- **App** – az alkalmazás fő logikája, eseménykezelés
  - **App.CreateEnv** – a globális állapot inicializálása, GUI funkcionalitás nélkül
  - **App.RunApp** – a főprogram definiálása, a main loop terminálásakor végrehajtandó IO akciók megadása
  - **App.Setup** – funkcionalitás hozzárendelése a GUI-hoz
  - **App.Types** – a globális állapothoz tartozó típusdefiníciók
- **Eval** – kifejezések GHCi-ban történő kiértékelése
  - **Eval.EvalMain** – a tényleges kiértékelést végző szál főprogramja
  - **Eval.Ghci** – az App számára biztosított interfész a kiértékeléshez
- **Persistence** – az App számára biztosított interfész fájlok mentéséhez és betöltéséhez

- **Spreadsheet** – a számolótábla reprezentációja és műveletei
  - **Spreadsheet.CodeGeneration** – kódgenerálás a kiértékeléshez, **FULL NEM ITT KÉNE LENNIE 44!4!!4**
  - **Spreadsheet.Interface** – a számolótábla műveletei, amiket az App használhat
  - **Spreadsheet.Parser** – felhasználó által írt kód reprezentációjának előállítása
  - **Spreadsheet.Types** – a számolótábla és kapcsolódó kivételek típusdefiníciói

3.3 A program komponenseinek részletes leírása

3.3.1 Spreadsheet.Types

Az alkalmazás a Spreadsheet típussal reprezentálja a számolótábla állapotát. Alább látható a Spreadsheet.Types modulban szereplő definíció:

```
type CellID = Int

data Cell' = Str String | Number Double | EmptyCell
  deriving (Eq, Show, Generic)

-- I need to come up with a better name lol
data ForPiece = Code String | Refs [CellID]
  deriving (Eq, Show, Generic)

data FormulaError = FNoParse
                  | FCycleRefError
                  | FNoCache
                  | FListTypeError
                  | FMissingDepError
                  | FGhciError
                  | FTimeoutError
  deriving (Eq, Show, Generic)

data Formula = Formula { _code :: String
```

```
20                          , _cache :: Either FormulaError Cell'
21                          , _value :: Maybe [ForPiece]
22                          }
23   deriving (Eq, Show, Generic)
24
25 data Cell = Val {_cellV :: Cell'} | For {_cellF :: Formula}
26   deriving (Eq, Show, Generic)
27
28 data Spreadsheet = SS { _sheet :: Gr Cell Int
29                        , _selected :: Maybe CellID
30                        , _logMessage :: Maybe String
31                        }
32   deriving(Eq, Show, Generic)
```

Code 3.2: A Spreadsheet típus

A Spreadsheet egy rekord típus, amelynek három mezője van. A *selected* mező jelenti az aktuálisan kijelölt cellát. Ez a mező kerülhetett volna a globális állapotba is, azonban a tervezés korai fázisában másképp döntöttem, és már nem feltétlenül éri meg refaktorálni a kódot. A *_logMessage* mező tartalmazza a legutóbbi művelet kiértékeléséből származó szöveges (a GUI-ban a logra írandó) üzenetet.

A *_sheet* mező reprezentálja a tényleges számolótáblát. A számolótábla egy irányított gráf, aminek a csúcsai *Cell* típusú értékekkel vannak címkézve. Az élek egész számokkal vannak címkézve. (Lehetett volna *()*-tal is, azonban a használt gráfcsomag által biztosított legrövidebb utak implementációnak szüksége volt számszerű élcímkékre. Az implementációban minden él címkéje *1.*) A gráfban minden csúcs a számolótábla egy cellájának felel meg. Egy A csúcsból pontosan akkor megy él egy B csúcsba, ha a B csúcsban található cella kódja hivatkozik az A csúcsban található cellára.

Egy cella pontosan akkor szerepel a gráfban, ha nemüres vagy van olyan cella, amelyik hivatkozik rá. Így az üres és nemhivatkozott cellák tárolására nincs szükség.

A fent megadott gráfreprezentációnak két további előnye is van. Egyrészt könnyű körfigyelést implementálni, így elkerülve, hogy cellák körkörösen hivatkozzanak egymásra; másrészt ha módosul egy A cella tartalma, akkor pontosan az A-ból elérhető csúcsoknak megfelelő cellákat kell újra kiértékelni.

A gráfreprezentáció megvalósításához az *fgl* csomagot használtam. A műveletek a *DynGraph* típusosztály tetszőleges megvalósítására működnek. **EZ MÉG NEM IGAZ, DE MAJD ÁTÍROM** A *Spreadsheet* típus definíciójában a PatriciaTree alapú *Gr* típust használtam.

Egy cella tartalmát a *Cell* típus fejezi ki. Egy cella tartalma lehet érték (*Cell'*) vagy formula (*Formula*). Az érték jelenleg háromféle lehet: szám (*Double*), string vagy üres.

Ha egy cella formulát tartalmaz, az a háromelemű *Formula* rekorddal reprezentáltatik. A *_code* mező tartalmazza a felhasználó által megadott kódot. Ez egy kényelmi funkció, hogy a kód megjelenítéséhez ne kelljen visszakonvertálni a reprezentációból. A *_cache* mezőben szerepel, hogy mi a formula legutóbbi kiértékelésének eredménye (ha egyáltalán már ki lett értékelve). A cache értéke vagy egy érték (*Cell'*) vagy valamilyen hiba (*FormulaError*).

A *_value* jelenti a formula kódgeneráláshoz szükséges reprezentációját. Ez a reprezentáció *ForPiece*-ek (formuladarabok) listája. Egy formuladarab vagy egy kódrészlet (*String*) vagy cellaazonosítók listája. A *_value* mezőről részletesebben lesz szó a *Spreadsheet.Parser* és a *Spreadsheet.CodeGeneration* modulok leírásában.

A *Formula* típushoz tartozik egy invariáns állítás: a program futása során egy *Formula* mindig a következő oldalon leírt állapotok valamelyikében figyelhető meg.

Érdemes megjegyezni, hogy ez az invariáns típuszinten is garantálható lett volna (feladat az olvasó számára!). A jelenlegi megoldás a korai tervezési fázis eredménye, a későbbiekben már erőforrásigényes lett volna refaktorálni a kódot.

További megjegyzések a *Spreadsheet* típussal kapcsolatban:

- A *Spreadsheet.Types* modul alapértelmezett nevű lenseket is exportál a *Cell*, *Formula* és *Spreadsheet* típusokhoz.
- A *Spreadsheet.Types* modulban szereplő összes típus (a kivételek kivételével) példánya a *Generic* és *Serialize* típusosztályoknak (ez utóbbit a *cereal* csomag exportálja). Erre a perzisztencia implementációjához van szükség.

Minta				Jelentés
Formula	_	(Left FNoParse)	Nothing	parseolási hiba
Formula	_	(Left FCycleRefError)	Nothing	sikeres parseolás, azonban a formula körkörös referenciákat adott volna a táblához
Formula	_	(Left FNoCache)	(Just _)	sikeres parseolás, érvényes referenciák, de a formula még nem lett kiértékelve
Formula	_	(Left FListTypeError)	(Just _)	**ALMA?**
Formula	_	(Left FMissingDepError)	(Just _)	a formula nem értékelhető ki, mivel egy hivatkozott cella nem volt cache-elve.
Formula	_	(Left FGHCIError)	(Just _)	a formula egyéb okokból nem volt kiértékelhető (pl. típushiba, Haskell szintaxishiba)
Formula	_	(Left FTimeoutError)	(Just _)	időtúllépés miatt sikertelen kiértékelés, valószínűleg végtelen ciklus miatt
Formula	_	(Right cell')	(Just _)	sikeres kiértékelés, az eredmény cell'

táblázat 3.1: Egy *Formula* lehetséges állapotai

3.3.2 Spreadsheet.Parser

A modul feladata egy a felhasználó által egy cellához megadott kód (*String*) reprezentációjának (*Cell*) előállítása. A modul egy függvényt exportál. (*rep :: String -> Cell)*

Legyen a felhasználó által megadott kód *str*. A *rep* függvény az alábbi specifikáció szerint állítja elő a kód cellareprezentációját.

1. Ha str = "", rep str = *Val EmptyCell*
2. Ha str illeszkedik a $(+|-|\varepsilon)DD^*(.D^+|\varepsilon))$ reguláris kifejezésre, ahol D=(0|1|2|3|4|5|6|7|8|9), akkor rep str = *Val (Num n)*, ahol n a literál által ábrázolt lebegőpontos szám.
3. Ha a fenti esetek egyike sem áll fent, és str nem (=C*) alakú (ahol C az összes karakterek halmaza), akkor rep str = *Val (Str str)*
4. Ha B a betűk halmaza (a *Data.Char* modul *isLetter* függvényének igazsághalmaza), C' = $C\backslash\{\S\}$ és str (= $((\S BD^+ : BD^+\S)|(\S BD^+\S)|(C'^+))^+$) alakú, akkor a kód formulaként parseolható. rep str = *Formula str (Left FNoCache) (Just ps)*, ahol ps definíciója alább szerepel.
5. Ha egyik fenti eset sem áll fent, akkor a parseolás sikertelen. Ekkor rep str = *Formula str (Left FNoParse) Nothing*

Ha *str* formulaként parseolható (fenti 4. eset), egy egyszerű szintaktikus elemzés segítségével kaphatjuk a reprezentációjának _*value* komponensét. A parser először elhagyja az = karaktert. Ezután sorban parseol substringeket a szó elejéről az alábbi módon:

1. Először megpróbálja cellahivatkozásként olvasni a soron következő részt: $((\S BD^* : BD^*\S)|(\S BD^+\S))$. Ha sikerült, a hivatkozást cellaazonosítók sorozatává konvertálja (lásd alább), és a kapott rs azonosítólistát $Refs\ rs$ módon az eredménylista végére fűzi.
2. Ha a soron következő substring nem olvasható cellahivatkozásként, akkor a parser végigolvassa a lehető leghosszabb $s = C'^+$ substringet, és az eredménylistához egy *Code* s-t ír.

A cellahivatkozások feloldásához kihasználjuk, hogy a karakterek injektíven az egész számok halmazára képezhetők (az *Enum* típusosztály műveleteivel). A kis- és nagybetűket nem különbözetjük meg. Emellett definiálunk egy *Enum* példányt *(Int,Int)* párokra. **AZT IS LE KÉNE SZÉPEN ÍRNI...** Jelölje $fromEnum^C$ a karaktert Int-té kódoló függvényt, és $fromEnum^P$ az *(Int,Int)* párt Int-té kódoló függvény. Jelölje $a, b :: Int$ esetén $[a..b]$ az $a \leq n \leq b$ n egész számok rendezett listáját! Ekkor a cellahivatkozások feloldása az alábbiak szerint történik:

1. Ha a hivatkozás ($\S BD^*\S$) alakú, legyen b a betű, és n a számjegyek által reprezentált egész szám. Ekkor a kapott rs lista egyelemű. $rs = [fromEnum^P\ (fromEnum^C, n)]$

2. Ha a hivatkozás ($\S BD^* : BD^*\S$) alakú, felbontható a *:* mentén két egyszerű hivatkozásra. Legyenek a betűk d_1 és d_2, a számjegyek által reprezentált számok pedig rendre n_1 és n_2! Ekkor XD LOL

3.3.3 Spreadsheet.Interface

Ebben a modulban szerepelnek az *App* komponens számára elérhető, a *Spreadsheet* típushoz kapcsolódó függvények. Alább látható táblázatos formában a modul által exportált függvények feladata:

setCellState

A *setCellState* függvény feladata, hogy a megadott cellaazonosítóhoz tartozó csúcsban lévő cella állapotát a felhasználó által megadott *String*-nek megfelelően módosítsa, valamint felülírja a *_logMessage* mező tartalmát a művelet sikerességétől függően.

Ehhez szükség van a megadott *String Cell* reprezentációjára, amit a *Spreadsheet.Parser* modul által exportált *rep* függvény számít ki. A kapott reprezentáció alapján az alább leírtaknak megfelelően viselkedik a függvény:

1. Ellenőrzi, hogy a cella állapotának megváltoztatásával keletkeznék-e körkörös referencia. Pontosan akkor keletkeznék, ha a megváltoztatandó c azonosítójú cellához tartozó csúcsba bemenő összes él kitörlésével keletkezett gráfban van

Függvény	Típus	Feladat
emptySpreadSheet	Spreadsheet	üres számolótábla (0 csúcsú gráf, nincs kijelölt cella, nincs log üzenet)
getCellText	CellID -> Spreadsheet -> String	egy cellában megjelenítendő szöveg lekérdezése
getCellCode	getCellCode :: CellID -> Spreadsheet -> String	egy adott cellába legutóbb beírt kód lekérdezése
setCellState	CellID -> String -> Spreadsheet -> Spreadsheet	a megadott cella állapotának módosítása egy a felhasználó által megadott String alapján
cacheCell	CellID -> Either EvalError String -> Spreadsheet -> Spreadsheet	kiértékelés eredményének cachelése
getSelected	Spreadsheet -> Maybe CellID	kijelölt cella azonosítója
setSelected	CellID -> Spreadsheet -> Spreadsheet	kijelölt cella azonosítójának beállítása
getLogMessage	Spreadsheet -> String	legutóbbi log üzenet lekérdezése

táblázat 3.2: A *Spreadsheet.Interface* által exportált függvények

olyan n azonosítójú csúcs, hogy c kódja hivatkozik n-re és a gráfban már van $c \to n$ út. (Ez utóbbi feltétel azt jelenti, hogy az n cella értéke függ c értékétől.) Ezt a feltételt az *isLegal* függvény ellenőrzi. Érdemes megjegyezni, hogy ilyen hiba csak akkor fordulhat elő, ha a kapott reprezentációnk egy formula.

2. Amennyiben az *isLegal* eredménye *False*, a c csúcsban levő cella reprezentációja *For (Formula str' (Left FCycleRefError) Nothing)* lesz, ahol *str'* a paraméterként kapott *String*. A *_logMessage* mezőbe egy hibaüzenet kerül.

3. Amennyiben az *isLegal* függvény *True* eredményt ad, a gráfból kitöröltetik az összes c-be menő él, és új c-be menő élek kerülnek behúzásra a c kódja által referált celláknak megfelelő csúcsokból. (Ezeket a *references* függvény számolja a reprezentációból.) A *_logMessage* mező tartalma egy sikert jelző üzenet lesz.

ITT MÉG KÉNE ÍRNI TALÁN A FELHASZNÁLT SEGÉDFÜGGVÉNYEKRŐL?

cacheCell

3.3.4 Spreadsheet.CodeGeneration

3.3.5 Eval.EvalMain

A modul által exportált *evalMain* függvény biztosítja a kifejezések GHCi-ben való kiértékelését végző szál főprogramját. A szál az alábbiak szerint működik:

1. Várakozik, ameddig a globális állapot *evalControl* mezőjének *eCommand* változójába egy GHCi utasítást nem ír a fő szál.

2. Kiüríti a változót, és kiértékeli a kapott utasítást a GHCi-ben.

3. Ha egy megadott idő után nem ér véget a kiértékelés (jelenleg 1 másodperc), lekérdezi a GHCi folyamathoz tartozó PID-et, majd megkeresi annak a gyerekfolyamatát (*childPid*), és az *eResult* változóba *Left childPid*-et ír. Ezt a folyamatot aztán a fő szál fogja kilőni.

4. Amennyiben időben véget ér a kiértékelés, a GHCi által eredményül adott sorok *result* listáját *Right result* módon az *eResult* változóba írja.

A timeout utáni viselkedés bonyolultsága egy szerencsétlen helyzet eredménye. A magyarázathoz meg kell ismerni a *ghcid* csomag által biztosított GHCi interfészt. A *startGhci* függvény a dokumentáció alapján elindít egy GHCi háttérfolyamatot, amellyel innentől egy megadott szálról kell interaktálni (a megszakítást kivéve). A valóságban azonban azt tapasztaltam, hogy két folyamatot indít el, amelyek közül az egyik gyereke a másiknak.

Időtúllépés esetén meg kell állítani a háttérben futó számítást. Erre szolgálna az *interrupt* függvény, ami egy *SIGINT* jelzést küld a GHCi folyamatnak. A GHCi folyamat azonban bizonyos esetekben ezt kimaszkolja, ilyenkor a számítást nem lehetséges megszakítani. **IDE KÉNE LINK ERRŐL A DISKURZUSRÓL** A GHCi leállítására szolgáló *stopGhci* függvény pedig csak az egyik (a szülő) folyamatot terminálja a *startGhci* által indított két folyamatból. A másik folyamat pedig tovább folytatja a számítást. Az alkalmazás egy Haskell szálat tartalmazó verziójában ez kiéheztette az fő folyamatot.

Ezért van szükség arra, hogy a kiértékelés külön szálon fusson. Ugyanis időtúllépés esetén a fő szál, miután ütemezésre kerül, a kapott PID alapján a lehető legagresszívabban (*SIGKILL*) terminálja a második GHCi folyamatot. Ez a tapasztalat szerint az első folyamatnak is véget vet. Ezután új GHCi folyamat indítható.

A fenti megoldást a szükség szülte, és tapasztalatok alapján, gyakran próbálgatás útján állt össze. Nem ismert, hogy miért indít a *startGhci* két folyamatot. (Egy "rendes" GHCi folyamathoz például csak egy PID tartozik.) A megoldás az én számítógépemen, Ubuntu 20.04 LTS operációs rendszer mellett működött, de nincs rá garancia, hogy más Linux rendszer (vagy akár egy másik számítógép!) esetén működni fog. (A működés feltétele, hogy ütemezésre kerüljön a fő szál.) Ráadásul csak e miatt az interakció miatt kellett konkurrenciát adni az alkalmazáshoz. Valódi hatékonyságot ezzel nem nyertünk, hiszen az egyik szál mindig blokkolt állapotban lesz. (Mivel mindkét szál a fogyasztóként hozzá tartozó *MVar*-ra várakozik, ha éppen a másik szál dolgozik.)

A gyerekfolyamat megtalálásához a program rendszerhívást hajt végre, a *pgrep* parancsot használja *-P* kapcsolóval.

Függvény	Típus
execGhciCommand	String -> ReaderT EvalControl IO (Either EvalError String)
loadModules	ReaderT EvalControl IO ()

táblázat 3.3: Az *Eval.Ghci* által exportált függvények

3.3.6 Eval.Ghci

A modul fő feladata, hogy kiértékeljen egy GHCi parancsot, és az eredményt értelmezze. Emellett lehetőséget biztosít a modulok és keresési útvonalak újratöltésére a globális konfiguráció alapján (*EvalConfig*).

A kiértékelés folyamata a (nem exportált) *execG* függvényben van leírva, és az alábbi módon zajlik:

1. A paraméterként kapott parancs a *eCommand* változóba kerül.
2. Kiolvasásra kerül az eredmény az *eResult* változóból.
3. Ha az eredmény *Left pid*, a kapott PID-hez tartozó folyamat terminálásra kerül, és a kiértékelés eredménye *Left ETimeoutError*. Új GHCi folyamat indul, és a hivatkozása bekerül a globális állapot *evalConfig* mezőjének *eGhci* mezőjébe.
4. Ha az eredmény *Right result*, ez a kiértékelés eredménye. *result :: [String]* a GHCi által eredményül adott sorok listája.

Az exportált *execGhciCommand* függvény ezt a viselkedést egészíti ki egy extra ellenőrzéssel.

1. Ha az *execG* eredménye *Left ETimeoutError*, akkor ez az eredmény.
2. Ha az *execG* eredménye *Right results*:
   - Ha *results* üres, az eredmény *Right ""*
   - Ha *results* egyelemű, az eredmény *Right (head (results))*
   - Ha *results* több elemű, akkor a GHCi több sornyi eredményt adott vissza. Ezt a függvény hibának tekinti, és az eredmény *Left (EGhciError results)*

A *loadModules* akció beállítja a globális állapot *evalControl* mezőjének *eConfig* mezője alapján az elérési utakat, majd betölti a megadott modulokat. A korábban betöltött modulokat először kitölti (*:m*). Ilyenkor a felhasználó által közvetlenül a GHCi-ba megadott definíciók is elvesznek. Az akció betölti az *Empty* modult is, ami szükséges ahhoz, hogy működjék az € minta, amit a cellablokkok kezeléséhez biztosít a program.

fejezet 4

Összegzés

?appendixname? A

Szimulációs eredmények

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque facilisis in nibh auctor molestie. Donec porta tortor mauris. Cras in lacus in purus ultricies blandit. Proin dolor erat, pulvinar posuere orci ac, eleifend ultrices libero. Donec elementum et elit a ullamcorper. Nunc tincidunt, lorem et consectetur tincidunt, ante sapien scelerisque neque, eu bibendum felis augue non est. Maecenas nibh arcu, ultrices et libero id, egestas tempus mauris. Etiam iaculis dui nec augue venenatis, fermentum posuere justo congue. Nullam sit amet porttitor sem, at porttitor augue. Proin bibendum justo at ornare efficitur. Donec tempor turpis ligula, vitae viverra felis finibus eu. Curabitur sed libero ac urna condimentum gravida. Donec tincidunt neque sit amet neque luctus auctor vel eget tortor. Integer dignissim, urna ut lobortis volutpat, justo nunc convallis diam, sit amet vulputate erat eros eu velit. Mauris porttitor dictum ante, commodo facilisis ex suscipit sed.

Sed egestas dapibus nisl, vitae fringilla justo. Donec eget condimentum lectus, molestie mattis nunc. Nulla ac faucibus dui. Nullam a congue erat. Ut accumsan sed sapien quis porttitor. Ut pellentesque, est ac posuere pulvinar, tortor mauris fermentum nulla, sit amet fringilla sapien sapien quis velit. Integer accumsan placerat lorem, eu aliquam urna consectetur eget. In ligula orci, dignissim sed consequat ac, porta at metus. Phasellus ipsum tellus, molestie ut lacus tempus, rutrum convallis elit. Suspendisse arcu orci, luctus vitae ultricies quis, bibendum sed elit. Vivamus at sem maximus leo placerat gravida semper vel mi. Etiam hendrerit sed massa ut lacinia. Morbi varius libero odio, sit amet auctor nunc interdum sit amet.

Aenean non mauris accumsan, rutrum nisi non, porttitor enim. Maecenas vel

tortor ex. Proin vulputate tellus luctus egestas fermentum. In nec lobortis risus, sit amet tincidunt purus. Nam id turpis venenatis, vehicula nisl sed, ultricies nibh. Suspendisse in libero nec nisi tempor vestibulum. Integer eu dui congue enim venenatis lobortis. Donec sed elementum nunc. Nulla facilisi. Maecenas cursus id lorem et finibus. Sed fermentum molestie erat, nec tempor lorem facilisis cursus. In vel nulla id orci fringilla facilisis. Cras non bibendum odio, ac vestibulum ex. Donec turpis urna, tincidunt ut mi eu, finibus facilisis lorem. Praesent posuere nisl nec dui accumsan, sed interdum odio malesuada.

Bibliography

[1] O. J. Dahl, E. W. Dijkstra és C. A. R. Hoare, szerk. *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[2] Thomas H. Cormen és tsai. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[3] Glenn E. Krasner és Stephen T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". *J. Object Oriented Program.* 1.3 (1988. aug.), old. 26–49. ISSN: 0896-8438. URL: `http://dl.acm.org/citation.cfm?id=50757.50759`.

[4] E. Dijkstra. "Classics in Software Engineering". Szerk. Edward Nash Yourdon. Upper Saddle River, NJ, USA: Yourdon Press, 1979. Fej. Go to Statement Considered Harmful, old. 27–33. ISBN: 0-917072-14-6. URL: `http://dl.acm.org/citation.cfm?id=1241515.1241518`.

?listfigurename?

?listtablename?

List of Codes