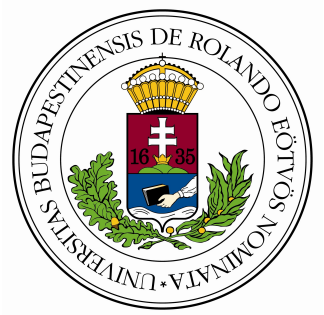


Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek

és Fordítóprogramok Tanszék

Interaktív program megvalósítása Agdában

Témavezetők:

Diviánszky Péter

tanársegéd

Páli Gábor János

tanársegéd

Szerző:

Hegedűs Dénes

Programtervező Informatikus BSc

Budapest, 2013

Tartalomjegyzék

1. Bevezetés

1.1. A feladat megfogalmazása

A szakdolgozat célja egy olyan egyszerű és jól használható interaktív program megvalósítása, ami bizonyos garanciákkal is rendelkezik. Ilyen elvárások lehetnek például, hogy a program ne kerülhessen végtelen ciklusba, vagy nem várt állapotba, ezen kívül bizonyos függvények helyes működését is szeretnénk garantálni.

1.2. Agda

Ezeknek az elvárásoknak a megvalósításához az Agda nevű tisztán funkcionális nyelvet használjuk fel, ami integrált tételbizonyító rendszerrel és erős típusrendszerrel rendelkezik, amelyek segítségével fordítás időben tudjuk biztosítani a következőket:

1. A program nem kerülhet végtelen ciklusba.
2. A programban nem léphet fel a programozó hibájából adódó futásidejű hiba.
3. A tesztelés része a programnak, ami szintén fordításidőben történik, így kizárólag olyan program állítható elő a fordítóval, amire a tesztesetek az elvárások szerint futnak le.

1.3. Funkcionális Reaktív Programozás

A program elkészítése során felhasználjuk a Funkcionális Reaktív Programozást (Functional Reactive Programming, FRP), aminek segítségével tisztán funkcionálisan, időtől függő folytonos függvények segítségével írhatunk le interaktív programokat. Ennek a modellnek az alapja a Funkcionális Reaktív Animáció (Fran) [7] amivel interaktív animációk készíthetők el.

Ennek megvalósításához az Agda nyelvben írt agda-frp-js függvénykönyvtárat használjuk fel, ami ECMAScript kimenetű, így böngészőben futtatható programot állít elő.

1.4. A programozási feladat

A megvalósítandó program a Mastermind nevű klasszikus táblajáték virtuális változata.

Ez egy kétfős kód-feltörő játék, amiben az egyik játékos a kód kitalálója, a másik pedig a kódfeltörő. A kód kitalálója megadott színek (6 féle) közül kiválaszt egy 4 elemű mintát, amit aztán a kódfeltörő minél kevesebb lépésben próbál megfejteni. A minta tartalmazhatja ugyanazt a színt többször is és a játék során a kódfeltörő számára rejtve marad. A próbálkozások számára felső korlát van (alapesetben 12), amit ha túllépünk, akkor a kód kitalálója nyert, ha pedig sikerül megfejteni, akkor a kódfeltörő.

A megvalósított programban mindig a számítógép a kód kitalálója, a játékos pedig a kód feltörője.

2. Felhasználói dokumentáció

2.1. Bevezetés

A felhasználói dokumentáció tartalmazza a program futtatásához és használatához szükséges információkat, első részében bemutatjuk a játék logikai szerkezetét, második részében pedig a felhasznált keretrendszert és a rendszerkövetelményeket.

2.2. Felhasznált módszerek

A program elkészítéséhez az Agda [2] nevű funkcionális nyelvet és az agda-frp-js [1] függvénykönyvtárat használtuk fel. Ez a függvénykönyvtár ECMAScript [6] kimenetet hoz létre, ami aztán kliens-oldali webes alkalmazásként webböngészőkben futtatható.

A felhasznált módszerek részletes leírásához lásd a Fejlesztői dokumentáció 3.3. fejezetét.

2.2.1. Agda

Az Agda [2] egy olyan funkcionális nyelv, ami integrált tételbizonyító rendszerrel rendelkezik, mely segítségével már fordítási időben tudjuk garantálni, hogy a program nem állhat le hibával, továbbá végtelen ciklusba se kerülhet. Emellett további állítások és megkötések is könnyen leírhatók benne, amikkel tetszőleges hibás állapotokat is elkerülhetünk, erős típusrendszere pedig kiszűri a típushibákat.

A program tesztelése a tételbizonyító rendszerrel már fordításidőben megvalósítható, ezzel biztosítva, hogy csak olyan programot tudunk lefordítani, amire a tesztesetek megfelelően működnek.

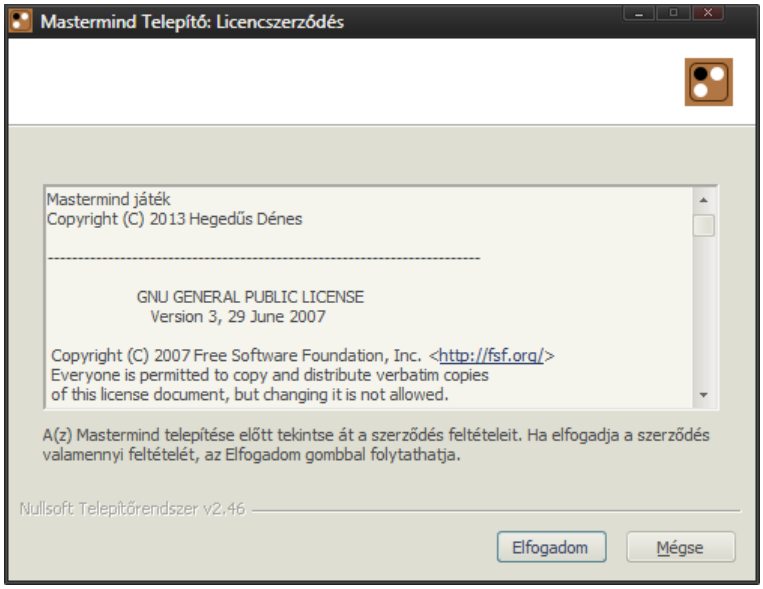
2.2.2. agda-frp-js

Az agda-frp-js [1] egy nyílt forráskódú függvénykönyvtár, aminek segítségével reaktív funkcionális programok írhatók le. A reaktív funkcionális programozás (Functional Reactive Programming, FRP) egy egyre inkább terjedő módszer interaktív

programok készítésére, amellyel időtől függő függvényekkel írhatunk le interaktív programokat.

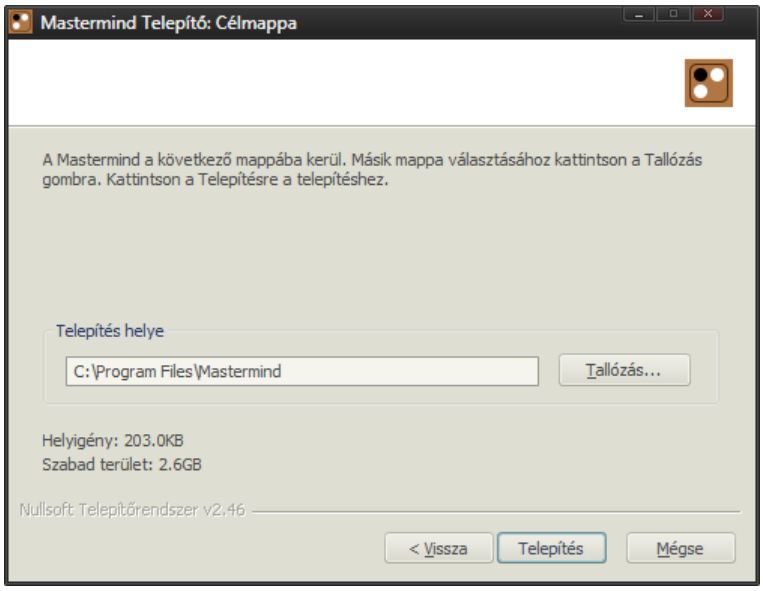
2.3. Telepítő

A programhoz Windows operációs rendszerhez telepítő tartozik. Ennek segítségével feltelepíthetjük a szoftvert egy tetszőleges mappába, amihez Start menübeli parancsikon és uninstaller is létrehozásra kerül. A telepítő 3 lépésből áll, az első ablakban a programról látható információ, az Elfogadom gombbal tudunk tovább lépni.



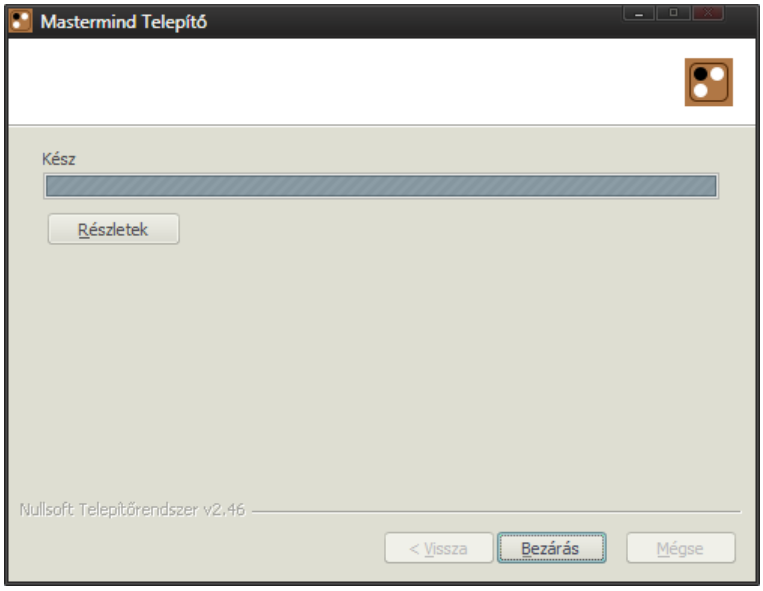
1. ábra. Telepítő

Ezután megkérdi az installer a telepítés helyét, ezután a Telepítés gombra kattintva elindul a telepítés, a Start menübeli parancsikon létrehozása és a Registrybe való felvétel.



2. ábra. Telepítő - Telepítés helye

A telepítés végeztével megtekinthetjük a felmásolt fájlokat vagy bezárhatjuk a telepítőt:



3. ábra. Telepítő - Vége

2.4. Forrásból való fordítás

A forrásból való fordításhoz szükséges az Agda 2.3.0.1-es verziója, és az agda-frp-js csomag lefordított változata, ennek megadásával Makefile segítségével (GNU Make

3.81) tudunk fordítani a forrás gyökérkönyvtárában kiadott paranccsal:

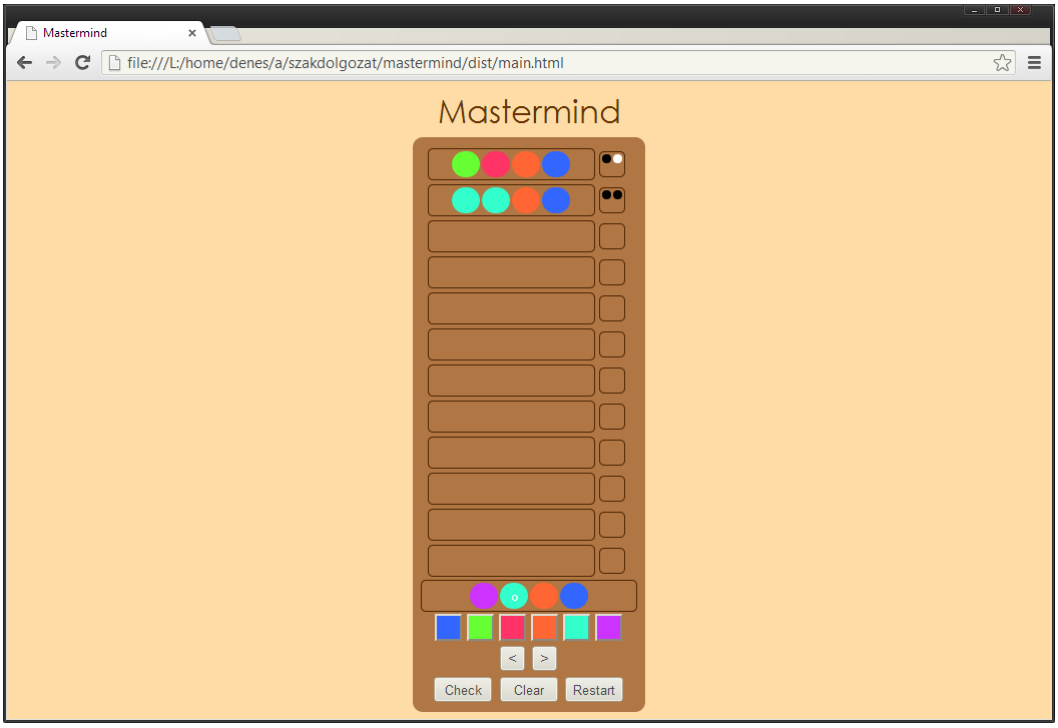
```
make FRP_DIR=<agda-frp-js helye>
```

Ez a parancs a dist mappába fordítja és másolja a kellő fájlokat, ezután a program futtatható is.

2.5. Futtatás

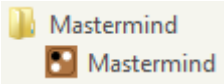
Az lefordított kódhoz egy egyszerű HTML keretet hozunk létre, ami a `main.html` fájlban található, a stílusát pedig a `mastermind.css` nevű stíluslapban írtuk le.

A játék a <Telepítés helye>/*main.html* megnyitásával indítható.



4. ábra. A program futás közben

Windows operációs rendszeren a telepítővel feltelepített program a Start menüben létrehozott parancsikonnal is indítható:

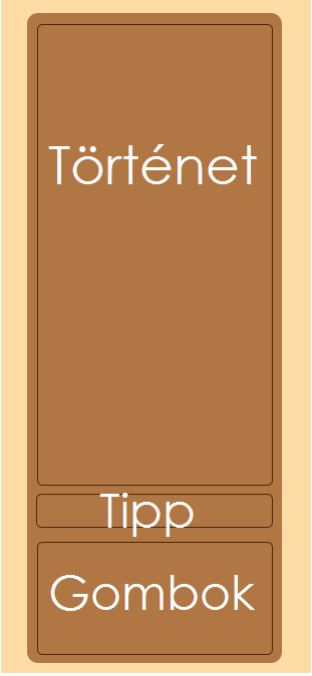


5. ábra. Start menü

2.6. A program működése

A program kinézetre világos hátterű, ezen belül a sötétebb keretben helyezkedik el az interaktív része, ami több részből áll:

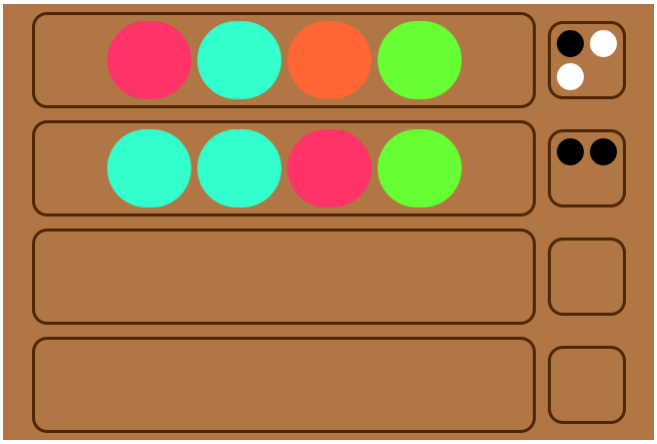
1. Felül, a keret körülbelül 2/3-át elfoglaló rész mutatja a korábbi próbálkozásokat, ez 12 sorból áll.
2. Alatta a tipp látható, ami 1 sor és szélesebb a történetnél.
3. A tipp alatt, miután véget ért a játék egy felirat lesz látható, ami jelzi, hogy nyertünk-e vagy vesztettünk.
4. Legalul a játékban való lépést és egyéb interakciókat megvalósító gombok helyezkednek el.



6. ábra. Elrendezés

2.6.1. Korábbi próbálkozások

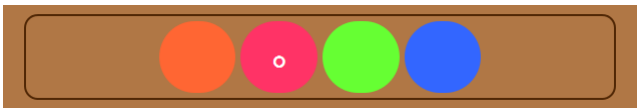
A korábbi próbálkozások listája kezdetben üres sorokat tartalmaz, ami a játék során felülről lefelé töltődik. Egy sor bal oldalán a tipp látható, mellette pedig sorrendtől függetlenül a fekete pöttyök olyan elemeket jelölnek, amik a tippben és a megoldásban színre és helyre is megegyeznek, a fehérek pedig olyan színeket, amik a tippben és a megoldásban is megtalálhatók, viszont más pozíción.



7. ábra. A történet

2.6.2. A tipp

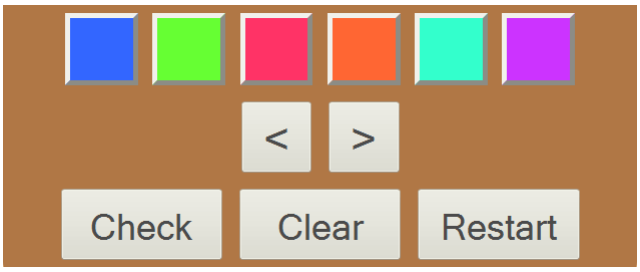
A korábbi próbálkozások alatt helyezkedik el az aktuális tipp, aminek változtatható mezőjét belül fehér kör jelöli.



8. ábra. A tipp

2.6.3. A gombok

A tipp után következnek a gombok:



9. ábra. A gombok

1. A színes gombokkal változtathatjuk az aktuálisnak megjelölt mező színét.
2. A '<' és '>' feliratúakkal tudjuk léptetni az aktuális mezőt.
3. A 'Check' gombbal tudjuk ellenőriztetni a tippet.
4. A 'Clear' gomb visszaállítja a tipp mintáját alapértelmezett (kék, kék, kék, kék) mintára.
5. A 'Restart' gomb pedig új játékot kezd.

2.6.4. A játék vége

A játék kétféleképp érhet véget, nyerés esetén „You won! :)" felirat látható, vesztés esetén pedig „You lost! :(Solution: [abcd]", ahol [abcd] jelöli a megoldásban lévő színek indexét.

2.7. Rendszerkövetelmények

A program tetszőleges operációs rendszeren, HTML5-t támogató, JavaScriptet futtatni képes webböngészőben futtatható, a következő operációs rendszerekben és böngészőkben teszteltem:

- GNU/Linux (Debian)
  - Mozilla Firefox 20.0
- Windows
  - Mozilla Firefox 20.0.1

– Google Chrome 26.0.1410.64

Internet Explorer esetén a böngésző JavaScript motorjának hiányosságai miatt a program nem futtatható.

3. Fejlesztői dokumentáció

3.1. Bevezetés

Fő célunk egy letisztult, rövid és egyszerű program megvalósítása, ami a felhasznált eszközökből kifolyólag olyan plusz garanciákat is ad, ami más nyelveken nem, vagy csak nehezen valósítható meg.

Ehhez felhasználjuk az Agda nevű tisztán funkcionális nyelvet és a hozzá megírt agda-frp-js csomagot, aminek segítségével ECMAScript kimenetű, böngészőben futtatható interaktív programot állítunk elő.

Emellett a fordítás során szeretnénk tesztelni a program működését, így további függvényekként egy külön modulban teszteseteket és további állításokat fogalmazunk meg a programról, amiket csak akkor fogad el az Agda típusellenőrzője, ha teljesülnek, ezután ezeket nem használjuk fel a program futása során, azonban lefordulásuk előfeltétele lesz annak, hogy a program maga is lefordúljon.

A megvalósított program a Mastermind nevű játék, amivel az interakció gombokon keresztül fog történni.

3.2. Elvárások

Legfőbb elvárásunk a programmal szemben, hogy típusai minél speciálisabbak legyenek, így a rajtuk alkalmazott függvények nem vihetik át a programot hibás vagy nem várt állapotba, továbbá szeretnénk bizonyításokkal alátámasztani a függvények helyes működését.

A programot fontos, hogy több, kisebb modulra bontsuk, amik jól különválaszthatóak és egyfajta hierarchia építhető fel rajtuk, így a különböző részei külön tesztelhetők. Mivel a megvalósítandó program interaktív, így célszerű különválasztanunk a megjelenítést a játék logikájától, ahol a nézet importálja a logikát. Emellett a program tesztelését is szeretnénk egy olyan modulban megadni, aminek a fordítása közben értékelődnek ki a tesztesetek. Ez a modul a modell és a nézet függvényeit is

felhasználhatja, de nem állíthat elő használható kódot.

A megvalósítás során hatékonysági szempontokat is figyelembe kell vennünk, mivel tisztán funkcionális programok esetén alkalmazott többszörösen egymásba ágyazott rekurziók lassú futást és nagy memóriaigényt eredményezhetnek.

3.3. Felhasznált módszerek

Ebben a részben bemutatjuk az Agda főbb nyelvi elemeit és a felhasznált függvénykönyvtár alacsony szintű függvényeit.

3.3.1. Agda

Az Agda egy függő típusos tisztán funkcionális nyelv [9], amiben Unicode [11] karaktereket is felhasználhatunk. A függvények megadása hasonló a matematikában alkalmazotthoz, a típusokat szinte minden esetben ki kell írnunk, a függvény paramétereinél a zárójelek pedig elhagyhatók, így például egy függvény, ami összead két természetes számot hasonlóképp néz ki:

```
f : ℕ → ℕ → ℕ
f x y = x + y
```

Típusokat konstruktoraik megadásával írhatunk le, amik tetszőleges paraméterszámúak lehetnek, például egy két elemű típus megadása:

```
data KetElem : Set where
  elem₁ : KetElem
  elem₂ : KetElem
```

A típusok lehetnek paraméteresek és függhetnek konkrét értékektől (függő típus) is, például az `A` típusú elemeket tartalmazó `n` hosszú vektorok egy definíciója:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A 0
  _::_ : ∀ {n : ℕ} (x : A) (xs : Vec A n) → Vec A (n + 1)
  -- üres vektor ([ ]) 0 hosszú
  -- x-et egy n hosszú xs vektor elé fűzve (x :: xs) n + 1 hosszú vektor lesz
```

Itt látható egy `{n : ℕ}` jelölés is, ez a kapcsos zárójel rejtett paramétert jelöl, amit a függvény törzsekben nem kell kiírnunk paraméterként akkor, amikor a fordító ki tudja azt következtetni.

Az Agda nyelvben továbbá kizárólag totális függvények adhatók meg, azaz például a funkcionális programozásból ismert `head` függvény, ami egy tetszőleges vektor első elemét adná vissza, nem adható meg:

```
head : ∀ {n : ℕ} {A : Set} → Vec A n → A
head (x :: xs) = x
head []        = {!!}   -- hiba
```

Itt üres vektor esetén egy olyan elemet kellene megadnunk, ami tetszőleges `A` halmaznak eleme, azonban ilyen Agdában nem lehetséges, így ilyen típusú függvényt nem tudunk megadni.

Ennek a tulajdonságnak köszönhetően nem tud a program „elszállni” és nem tudunk a program futása során kivételeket sem dobni, hanem minden függvénynek át kell vinnie a programot egy másik állapotba, ezzel sok hibás esetet kiszűrve.

A nyelv rendelkezik továbbá mixfix operátorokkal, névtelen (lambda) függvényekkel és kötési erősség megadásával is, ezekre egy-egy példa:

```
  -- függvénykompozíció
_∘_ : ∀ {A B C : Set} (f : B → C) → (g : A → B) → (A → C)
f ∘ g = λ x → f (g x)

  -- balról zárójeleződő, 3-as precedenciájú operátor
infixl 3 _∘_
```

Megadhatók a nyelvben olyan függvények és értékek is, amiknek típusuk van, de Agda-beli definíció nem tartozik hozzájuk, ezeket a `postulate` kulcsszó segítségével tudjuk bevezetni. Ennek használata kerülendő, mivel így tetszőleges típushoz tudunk új elemeket megadni, amivel kijátszható a típusellenőrzés.

3.3.2. Funkcionális Reaktív Programozás

A felhasznált agda-frp-js függvénykönyvtár a Funkcionális Reaktív Programozás (Functional Reactive Programming - FRP) egy modelljének [8] implementációját tartalmazza. Ebben a modellben időtől függő mellékhatásmentes függvények segítségével adható meg a programmal való interakció.

A függvénykönyvtár tartalmazza a modellt, az interakciót segítő magasabb szintű függvényeket és az ECMAScript kimenetet előállító részeket is, moduljainak elnevezése `FRP.JS.`*modulnév*.

Ebben a modellben a legfőbb típusok a reaktív típusok, amik idő szerint változnak, ennek definícióját a könyvtárban (module FRP.JS.RSet) egyszerű típusszinonimaként adták meg:

```
RSet : Set₁
RSet = Time → Set
```

Azaz `RSet` egy olyan típus, ami minden időpillanathoz egy másik típust rendel, ezután további segéd típusok következnek.

A ⇒ operátor segítségével reaktív típusok közti függvényképzést ad meg, konstans reaktív típusra pedig a ⟨`A`⟩ jelölést alkalmazza (ez minden időpillanathoz az `A` halmazt rendeli).

```
_⇒_ : RSet → RSet → RSet
(A ⇒ B) t = A t → B t

infixr 1 _⇒_

⟨_⟩ : Set → RSet
⟨ A ⟩ t = A

⟦_⟧ : RSet → Set
⟦ A ⟧ = ∀ {t} → A t
```

Fontos jelölés továbbá a ⟦ `A` ⟧ is, ami azon függvények típusa, amik minden t időponthoz hozzárendelnek egy `A t` halmazbeli elemet. Erre egy példa:

Legyenek t_1 és t_2 időpontok, amiken legyen definiálva egy rendezés:

```
postulate
  t₁ t₂ : Time
  _<_ : Time → Time → Bool
```

Legyen `A` egy reaktív (azaz időtől függő) típus, például legyen `A` t_1 és t_2 között `Bool`, egyébként pedig $\mathbb{N}$.

```
A : RSet
A t with (t₁ < t), (t < t₂)   -- A t
... | true, true = Bool       -- | t₁ < t < t₂ = Bool
... | _ = ℕ                   -- | egyébként = ℕ
```

Ekkor $[\![$ `A` $]\!]$ azon függvények típusa, amik minden t időponthoz hozzárendelnek egy `A t` beli értéket, azaz például egy függvény, aminek értéke t_1 és t_2 között false, egyébként pedig egy természetes szám:

```
f : [[ A ]]
f {t} with (t₁ < t), (t < t₂)   -- f t
... | false, _    = 3           -- | t < t₁ = 3
... | true, true  = false       -- | t₁ < t < t₂ = false
... | true, false = 12          -- | t₂ < t = 12
```

A függvénykönyvtárban a további alacsony szintű típusok és függvények posztulátumként vannak megadva, ezekből a fordító számára külön megadott, kész ECMAScript kódokból generálódik a futtatható kód. Ennek jelölése egy f függvényre:

```
{-# COMPILED_JS f <ECMAScript kód> #-}
```

Ilyen típusok a `Beh` (Behaviour modul), `Evt` (Event modul), `DOM` és a hozzájuk tartozó függvények, a Behaviourök segítségével állapotátmenetek, Eventekkel az interakció valósítható meg, a `DOM` modulban pedig a HTML elemeket felépítő és módosító függvényeket definiálták, ezekből a következőket használjuk fel:

```
module Event where
  postulate
```

```
    Evt : RSet → RSet
    accumBy : ∀ {A B} → ⟦ ⟨ B ⟩ ⇒ A ⇒ ⟨ B ⟩ ⟧ → B →
                ⟦ Evt A ⇒ Evt ⟨ B ⟩ ⟧
  module Behaviour where
    open Event
    postulate
      Beh : RSet → RSet
      map : ∀ {A B} → ⟦ A ⇒ B ⟧ → ⟦ Beh A ⇒ Beh B ⟧
      [_] : ∀ {A} → A → ⟦ Beh ⟨ A ⟩ ⟧
      hold : ∀ {A} → ⟦ ⟨ A ⟩ ⇒ Evt ⟨ A ⟩ ⇒ Beh ⟨ A ⟩ ⟧
```

A `[_]` jelöléssel adhatunk meg konstans Behaviour értékeket, például ilyenek lesznek a programban a gombok feliratai és a HTML elemek stílusa.

Később, hogy ne kelljen a programban ilyen alacsony szintű típusokat és függvényeket használnunk, a Mastermind.View.Base modulban típusszinonimákkal elrejtjük ezeket a típusokat, így a program fő része áttekinthetőbb lesz.

Ezek után definiálták a Behaviour modulban a program működését biztosító, állapoton való léptetést biztosító függvényt, ezt accumHoldBy-nak nevezték el.

```
accumHoldBy : ∀ {A B} → ⟦ (⟨ B ⟩ ⇒ A ⇒ ⟨ B ⟩) ⟧ → B →
                ⟦ Evt A ⇒ Beh ⟨ B ⟩ ⟧
accumHoldBy f b σ = hold b (accumBy f b σ)
```

Ennek első paramétere a léptetést megvalósító függvény, aminek szignatúrája `f : Allapot → Muvelet → Allapot`, második paramétere egy kezdeti állapot, a harmadik pedig a műveleteket tartalmazó esemény. A mi esetünkben az állapot egy rekord lesz, a műveletek pedig a gombok.

Az FRP.JS.DOM modulból a következő függvényeket használjuk majd:

- element „<HTML tag>” <elem>: statikus HTML elem
- text <szöveg>: szöveg hozzáadása

- attr „<HTML attribútum>” <attr>: attribútum hozzáadása az aktuális HTML elemhez

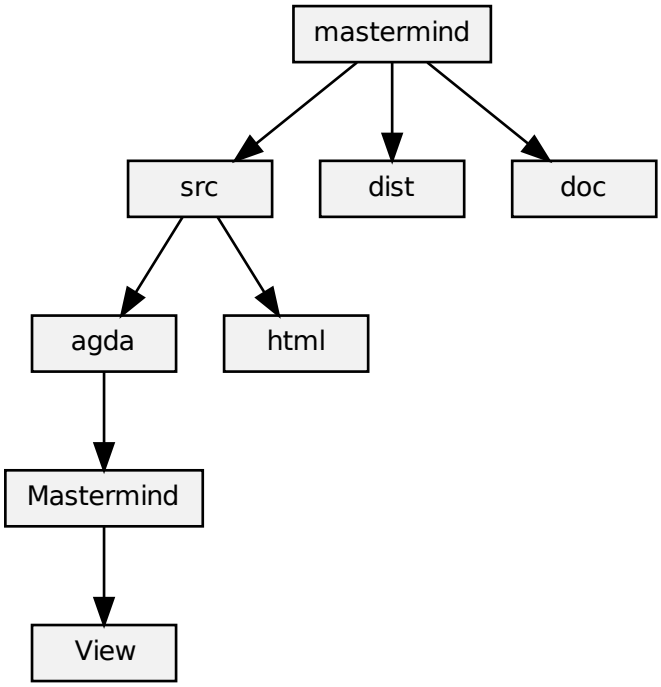
Ezeknek a függvényeknek van <függvénynév>+ nevű változataik is, ezekkel lehet az interakciót megvalósító elemek (például gombok) tulajdonságait hasonlóan megadni.

3.4. A program logikai és fizikai szerkezete

3.4.1. Mappaszerkezet

A program mappaszerkezete a következő:

- *dist*: A fordító által generált kimeneti fájlok mappája.
- *src*: A forrás, ezen belül az *agda* mappa tartalmazza a program forrását, a *html* pedig a HTML keretet és a stílust.
- *doc*: A dokumentáció.

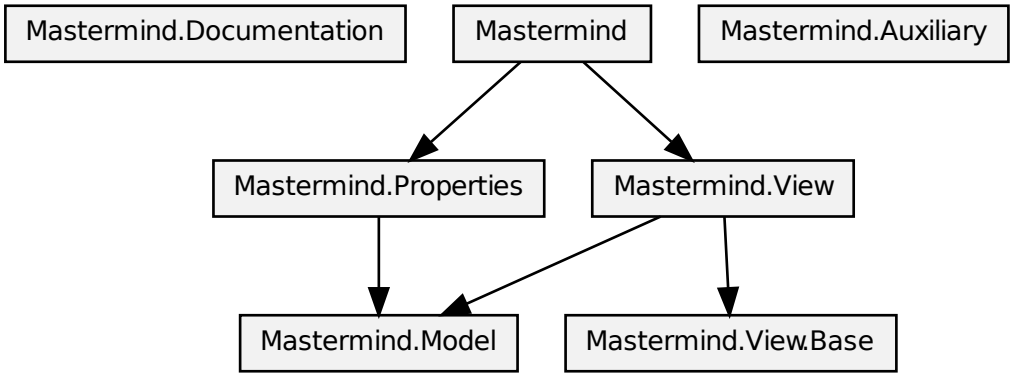


10. ábra. Mappák hierarchiája

A programhoz a gyökérkönyvtárban mellékeljük a *LICENSE* fájlban a programhoz tartozó licencet, itt található továbbá a fordítást segítő *Makefile* is, aminek használata a 2.4-es fejezetben olvasható.

3.4.2. Modulok

A programot a modell-nézet architektúra alapján építjük fel, azaz külön választjuk a megjelenést a logikai szerkezettől.



11. ábra. Modul hierarchia

A program kezdeti (`main`) függvénye a `Mastermind` modulban (`Mastermind.agda`) található, ezt a modult adjuk át a fordítónak. Ezután a modell-nézet architektúrának megfelelően különválasztjuk a megjelenést a logikai szerkezettől, a fő modul importálja a nézetet, amit a `Mastermind.View` (`Mastermind/View.agda`) modul implementál. A program logikai szerkezetét a `Mastermind.Model` (`Mastermind/Model.agda`) modulban helyeztük el, ebben találhatók az adatszerkezetek és az ezeket manipuláló függvények definíciói.

Az `agda-frp-js` függvénykönyvtár hátránya, hogy nem használnak az Agda standard library-ból semmilyen modult sem, és mi sem használhatjuk a kettőt együtt. Ennek hátránya az, hogy sok mindent nem vettek át a standard könyvtárból, ezért egy `Mastermind.Auxiliary` nevű modulban gyűjtöttem össze a hiányzó, de nekünk kellő függvényeket és típusokat.

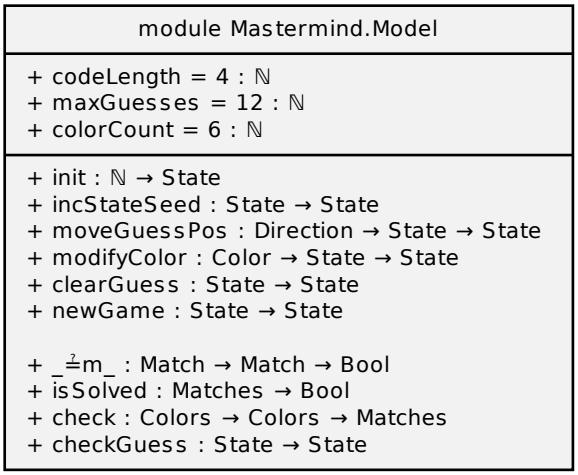
A program tesztelését és a függvényekre vonatkozó állításokat a `Mastermind.Properties` (`Mastermind/Properties.agda`) valósítja meg, ez importálja a nézetet és a modellt is, őt pedig a fő modul (`Mastermind`) importálja, de nem használ fel belőle semmilyen függvényt. Ennek segítségével a tesztelés fordítás időben megtörténik, de ez nem változtatja meg a program működését.

A szakdolgozat dokumentációja a `Mastermind.Documentation` (`Mastermind/Documentation.lagda`) modulban található, ami Literate Agda kiterjesztésű, azaz helyes LaTeX és Agda kódot vegyesen tartalmaz, ebből például az *lhs2TeX* nevű programmal tudunk *.tex* fájlt előállítani, amiből pedig a *pdflatex* parancs kiadásával elkészül a dokumentáció pdf formában. A dokumentációban található ábrák a *graphviz* programmal készültek.

A megjelenítést HTML [4] (`main.html`) és CSS [3] (`mastermind.css`) segítségével végezzük, ahol a `main.html` fájl betölti a lefordított ECMAScript kódokat egy keretbe, a stíluslap pedig a programban megadott HTML attribútumok (`class`, `id`) alapján színezi és igazítja az elemeket.

3.5. Modell

A Modell (module Mastermind.Model) tartalmazza az állapot rekordot (record State), az összes állapotot módosító függvényt és a véletlenszámgenerátort (module Random).



12. ábra. Modell Diagram

A modellben három konstans található, ezek a hagyományos Mastermind paraméterei, ahol a minta 4 színből áll (codeLength), 12-őt tippelhetünk összesen (maxGuesses) és 6 színből választhatunk (colorCount).

$$\mathsf{codeLength} = 4$$

$$\mathsf{maxGuesses} = 12$$

$$\mathsf{colorCount} = 6$$

3.5.1. Típusok

A modul néhány, pár elemű típust definiál, ezeket használva általánosabb típusok helyett jelentősen leszűkül az állapottér mérete és könnyebb az ezeket használó függvényekre állításokat is megfogalmazni. Ezek a típusok:

- `Match`: black, white, none - találatok jelölése

- `End`: won, lost - játék végének esetei
- `GameState`: active, ended won, ended lost - játék állapota
- `Direction`: $\leftarrow$, $\rightarrow$ - irány jelölése

Ezek után típusszinonimákat adunk meg, amik csak a program olvashatóságát javítják, funkcionális szerepük nincs:

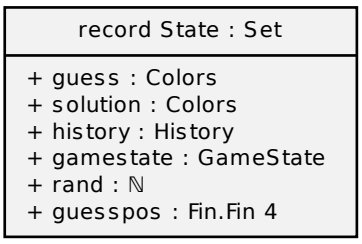
```
Color     = ℕ
Colors    = Vec Color codeLength
Matches   = List Match
HistElem  = Colors × Matches
History   = List HistElem
```

A színek (`Color`) típusa természetes szám, így a backendre bízzuk a színek megjelenítését (`mastermind.css`), az ilyen színekből álló, kódhossz hosszúságú vektorok típusa a `Colors`. A találatok listájának szinonimája a `Matches`, a színek vektorából és ilyen találatok listájából álló pár típusa pedig a `HistElem`, a történet (`History`) ilyen elemek listája.

3.5.2. Állapot

Az állapotot rekordként adjuk meg, a következőképp:

```
record State : Set where
  field
    guess     : Colors      -- aktuális tipp
    solution  : Colors      -- megoldás
    history   : History     -- történet
    gamestate : GameState   -- játék állapota
    rand      : ℕ           -- véletlen számok generálásához
                            -- szükséges kezdeti érték (seed)
    guesspos  : Fin.Fin 4   -- a tipp aktuálisan módosítható mezője
```



13. ábra. Állapot

Ezen kívül megadunk állapot-módosító függvényeket, amiket majd a nézetből használunk, ezek:

- `init`: kezdeti állapot, paramétere egy kezdeti érték a véletlenszám generátorhoz
- `incStateSeed`: a véletlenszám generátor léptetése, anélkül hogy felhasználnánk a generált értéket.
- `moveGuessPos`: a tipp aktuális mezőjén léptet a megadott irányba ($\leftarrow$, $\rightarrow$)
- `modifyColor`: az aktuális mezőn lévő szín változtatása
- `clearGuess`: kezdeti állapotba állítja a tippet (kék, kék, kék, kék)
- `newGame`: új játék kezdése

3.5.3. Véletlenszámok

Véletlenszámok generálásához szükséges egy modulo operátor, amit tisztán funkcionális nyelveken rekurzióval tudunk megadni, azonban itt a generálás során nagy számokkal használnánk, emiatt az Agda fordító a fordítás közbeni egyszerűsítések során nagy mértékben belassult, a generált ECMAScript kód pedig „too much recursion” hibaüzenettel leállt. A lassulás arra vezethető vissza, hogy az Agda fordító megpróbálja a megadott konstansokkal elvégezni a modulo műveletet, ezzel egyszerűsítve a függvényhívást a véletlenszám generátorban. Ennek megakadályozására

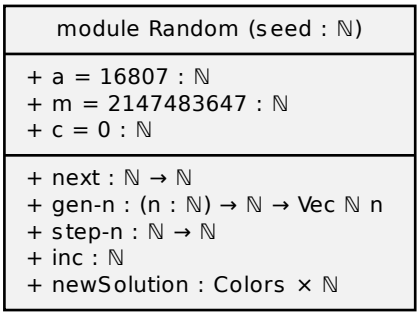
például az `abstract` kulcsszóval átláthatatlanná tehető a definíció, azonban ennek a használatával is fennállt a probléma.

A hiba kiküszöbölésére így a függvényt konstans 0-ként adjuk meg Agdában, a fordítónak pedig adunk egy hatékony JavaScript kódot használatra:

```
_%_ : ℕ → ℕ → ℕ
n % m = 0
```

```
{-# COMPILED_JS _%_ function(x)  return function(y)
if (x < 0)  return (x % y) + x;  else  return x % y;  ;  #-}
```

Ennek hátránya, hogy így állításokat és teszteseteket nem tudunk helyesen megadni a véletlenszám generálással kapcsolatban.



14. ábra. Véletlenszámok

Ezután következik a Random modul, ami egy paraméteres modul, paramétere a kezdeti érték (seed), mikor a program során használni szeretnénk, akkor az aktuális seed értékkel kell megnyitnunk a modult. Ez a modul egy úgynevezett Linear Congruential Generator (LCG) egy megvalósítása, Park és Miller „Minimal Standard" [10] változata alapján.

```
module Random (seed : ℕ) where
```

```
private
  a = 16807
  m = 2147483647
  c = 0
```

Egy LCG három konstanst tartalmaz, ezek segítségével generál elemeket a következő képlet alapján:

$$X_{n+1} = (a * X_n + c) \bmod m$$

ahol $X_0 = seed$.

```
next : (n : ℕ) → ℕ
next n = (((a * n) + c) % m)
```

Ezek segítségével a `newSolution` függvény generál egy új megoldást és visszaadja a megváltozott kezdeti értéket, amit aztán eltárolunk későbbi véletlen számok generálásához.

```
newSolution : Colors × ℕ
newSolution = mapv (λ x → (x % colorCount) + 1) (gen-n 4 seed), step-n 4
```

Itt a `gen-n` függvény `n` hosszú vektornyi véletlen számot állít elő, a `step-n` pedig `n`-et léptet a kezdeti értéken. A `newSolution` függvény mellett exportálunk egy `inc` nevű függvényt, amivel a program futása során majd minden gombnyomásra léptetünk egyet az aktuális seeden, így amikor új megoldás generálására van szükség, még véletlenebb értékeket kapunk.

```
inc : ℕ
inc = step-n 1
```

Mivel az Agda nyelv tisztán funkcionális, így külső értéktől nem függhetnek a függvények, azonban az agda-frp-js függvénykönyvtárban a programok egy rejtett idő paramétertől is függnek, amit elérünk kezdetben, ez jól használható kezdeti értéknek.

3.5.4. Tipp ellenőrzése

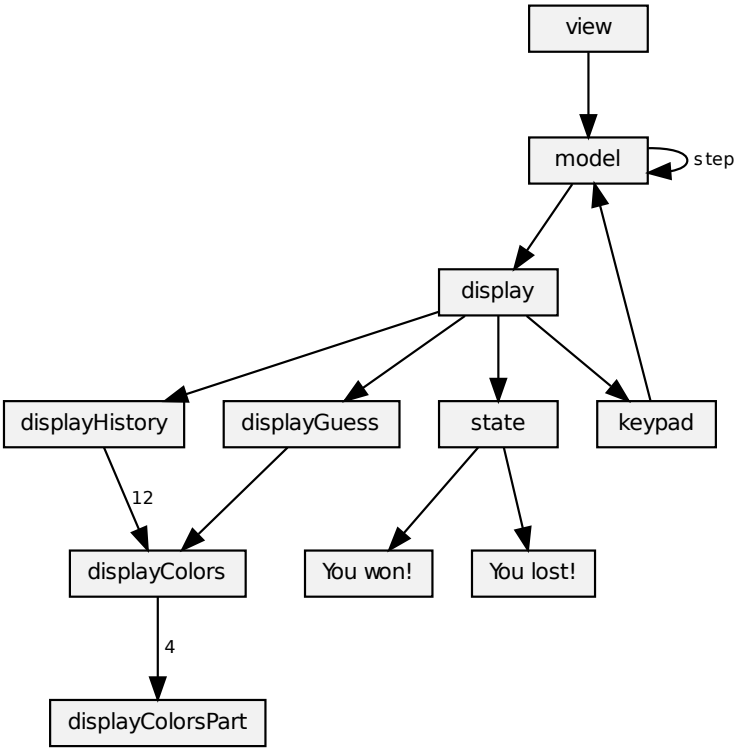
A felhasználó által megadott tippet a `check` függvénnyel ellenőrizzük. Ennek működése a következő:

```
check : Colors → Colors → Matches
check guess solution =
  blacks ++ whites
    where
      zipped = toList (zipv guess solution)
      diffs  = unzip (filter (¬ ∘ uncurry _≟_) zipped)
      rg     = proj₁ diffs
      rs     = proj₂ diffs
      whitec = length (rg \\ (rg \\ rs by _≟_) by _≟_)
      whites = replicate whitec white
      same   = filter (uncurry _≟_) zipped
      blacks = map (const black) same
```

Összefűzzük a tippet a megoldással (`zipv guess solution`), ezt listává alakítjuk (`toList`), így olyan listát kapunk (`zipped`) aminek minden eleme a tipp és a megoldás azonos pozíción lévő eleméből álló rendezett pár, ebből a fekete értékek (jó szín van jó helyen) száma egyenlő azon elemek számával, ahol a pár két eleme megegyezik (`same`). A fehér pöttyök (van olyan szín, de rossz helyen) esetén ezeket a mezőket eldobjuk (`diffs`), ezután a tippből maradt elemekből (`rg`) halmazként kivonva a megoldásból megmaradt elemeket (`rs`) megkapjuk a tipp azon elemeit, ami nem szerepelt a megoldásban. Ezután ezeket ismét halmazkivonással kivonva megkapjuk azokat az elemeket, amik szerepeltek, de rossz helyen, ezek száma (`whitec`) lesz a fehér pöttyök száma.

3.6. Nézet

A nézet tartalmazza a megjelenítéssel és az interakcióval kapcsolatos függvényeket. Ez a modul felhasználja a Modell által exportált állapot-módosító függvényeket és itt kezeljük a meg nem engedett eseteket is (pl. ha már vége a játéknak, akkor ne lehessen tippelni).



15. ábra. Megjelenítés

Ez a modul felhasznál egy `Mastermind.View.Base` nevű modult is, ebben típusszinonimákkal elrejtem az agda-frp-js által megadott időtől függő típusokat, ettől a kód átláthatóbb lesz, és itt találhatók a nézet által felhasznált általánosabb függvények is.

A nézet egyetlen adatszerkezete a gombokat írja le (`Button`):

```
data Button : Set where
```

```
color   : Color → Button      -- szín változtatása
move    : Direction → Button  -- léptetés a tippben
ok      : Button              -- tipp ellenőrzése
clear   : Button              -- tipp törlése
restart : Button              -- játék újrakezdése
```

Ezután következnek a típusszinonimák:

```
InteractiveButtons = ∀ {w} → ⟦ Beh (DOM w) ∧ Evt ⟨ Button ⟩ ⟧
ButtonHandler      = Button → InteractiveButtons
```

Interaktív gombnak (`InteractiveButtons`) nevezzük el azt a típust, ami tartalmazza a gombok kinézetét (`Beh (DOM w)`) és a gombokkal történő interakciót (`Evt ⟨ Button ⟩`) is.

A `ButtonHandler` típus elemei olyan függvények, amik az általunk definiált gombokra ilyen interaktív gombokat hoznak létre.

Ezután következnek a megjelenítést végző függvények:

```
UIElement = ∀ {w} → ⟦ Beh ⟨ State ⟩ ⇒ Beh (DOM w) ⟧
Model     = ⟦ Evt ⟨ Button ⟩ ⇒ Beh ⟨ State ⟩ ⟧
View      = ∀ {w} → ⟦ Beh (DOM w) ⟧
```

Az `UIElement` típus minden időpillanatra az aktuális állapotból készít egy megjeleníthető elemet (DOM), ezekből építjük fel az interfészt.

A `Model` gombokon értelmezett eseményekből állítja elő az állapot változását (`Behaviour`).

A `View` típus kész, kirajzolható HTML elemeket állít elő.

A nézet egyik fő függvénye a `step`, ez valósítja meg az állapotban való lépést, típusa: `State → Button → State`, azaz megkapja az aktuális állapotot és a lenyomott gombot, és ezekből egy új állapotot állít elő. Ebben a függvényben ellenőrizzük továbbá, hogy az adott gomb használható volt-e, mikor lenyomták (például ha vége a játéknak, akkor nem lehet többet tippelni), ez azért szükséges itt, mert a nézetben definiáltuk a gombokat is.

Ezután megadjuk egy `model` nevű függvényt, ami a felhasznált módszereknél ismertetett `accumHoldBy` függvényt felhasználva, a modell `init` függvényével előállít egy kezdeti állapotot, amit aztán a `step` függvénnyel léptet tovább. Ez egy reaktív típusú függvény, így elő tudjuk hozni a rejtett paraméterét (`epoch (<idő> ms)`), ami a program indításának időpontja, ezt adjuk át az `init` függvénynek, azaz ez lesz a véletlenszám generátor kezdeti értéke.

```
model : Model
model {epoch (time ms)} =
  accumHoldBy step (init time)
```

Interaktív gombokat a `button` függvénnyel hozunk létre, ami adott gombra előállítja az eseményt, a címkéjét és a stílusát.

```
button : ButtonHandler
button button =
  listen+ click (λ _ → button) (                -- esemény
    element+ "button" (                         -- HTML gomb
      text+ [button$ button] +++                -- felirat
      attr+ "class" [buttonClass button]))      -- HTML osztály
```

Az összes használható gombot a `keypad : InteractiveButtons` állítja elő, ez a függvény adja meg a sorrendjüket és az elhelyezkedésüket is.

Ezek után következik a nézet nem interaktív része, az itt lévő típusok az állapotot képzik le HTML elemekre, eseményeket nem állítanak elő, így típusukban a visszatérési érték a fentebb leírt `UIElement` típusú.

A `displayGuess : UIElement` függvény a tippet jeleníti meg, a `displayHistory : UIElement` pedig a teljes történetet, ennek mind a 12 sora 2 részből áll, az akkori tippből, és a rá kapott fekete/fehér pöttyökből.

A `display` függvény használatával állítjuk elő a nézet teljes nem interaktív részét, a történetből táblázatot képzünk, alatta helyezkedik el az aktuális tipp, utána pedig, ha véget ért a játék egy felirat lesz látható.

```
display : UIElement
display σ =
  element "table" (                    -- történet
    displayHistory maxGuesses σ ++
    attr "id" ["historytable"]) ++
  element "div" (displayGuess σ) ++    -- aktuális tipp
  element "div" (text (map state$ σ))  -- nyert/vesztett felirat
```

A teljes nézetet a `view` függvény állítja elő, ez létrehozza a nézetet az interaktív gombok eseményeivel (`display (model evt)`), utána pedig hozzáfűzi a gombok HTML elemeit. Ezt a függvényt használja fel a fő (`Mastermind`) modulban lévő belépési pont (`main`).

```
view : View
view with keypad
... | (dom, evt) = display (model evt) ++ dom
```

3.7. HTML és CSS

A lefordított kód betöltését a `main.html` nevű HTML fájl végzi, ebben egy egyszerű táblázat segítségével egy középre igazított keretbe töltődik be a program, ennek a kódja:

```
<div class="agda" id="mastermind" data-agda="Mastermind"></div>
```

A stíluslap (CSS) a (`mastermind.css`) fájlban található, ami többféle osztályt és azonosítót is használ, ezek a következők:

- Általános:
  - `.agda` - a teljes programra vonatkozó stílus
  - `#maintable` - a teljes képernyőt kitevő keret
  - `#maintitle` - „Mastermind” felirat kinézete
  - `#mainpart` - a program kerete

- Korábbi próbálkozásokra vonatkozó:
  - `#historytable` - a keret stílusa
  - `#matchtable` - a találatok kerete
  - `#hist_matches` - a találatok mérete
  - `#black_m`, `#white_m` - a találatok színe
- Aktuális tippre vonatkozó:
  - `#guesstable` - a keret stílusa
  - `.color` - minden színre vonatkozó értékek
  - `.color[1-6]` - színek hozzárendelése a programban használt reprezentációhoz ($\mathbb{N}$)
  - `#guess_act` - a tipp aktuális elemének megkülönböztetésére használt stílus
- Gombok:
  - `.button_num` - színek gombjai
  - `.button_op` - műveletek gombjai

3.8. Tesztelési terv

A szoftver teszteléséhez az Agda nyelv beépített tételbizonyító rendszerét használjuk fel, amivel a program futása során fennálló összefüggések mellett tesztesetek is megadhatók. Ezeket a `Mastermind.Properties` modulban definiáltuk, amit a legfelső (`Mastermind`) modul importál, így a tesztelés fordítás közben zajlik, ezzel garantálva, hogy csak helyesen működő programot lehessen lefordítani.

3.8.1. Nyelvi elemek

Az Agda nyelv szigorú típusrendszere és különféle nyelvi elemei segítségével elkerülhetjük a hibás állapotba lépést is már a fordítás során, ami a tesztelést is megkönnyíti. Ezek a nyelvi elemek minden függvényre/típusra adottak:

- *Totális függvények:* A függvény az állapotterének minden elemét le kell hogy képezze valamilyen elemre.
- *Terminálás ellenőrzés:* A függvények nem kerülhetnek végtelen ciklusba (minden rekurziónak struktálisan kisebbnek kell lennie az eredeti függvényhívásnál, stb.).

A teszteléshez felhasználjuk a propozícionális egyenlőséget (Propositional Equality) [5], amit a `Mastermind.Properties` modul elején definiálunk.

```
data _≡_ {A : Set} (a : A) : A → Set where
  refl : a ≡ a

infix 4 _≡_

data ⊥ : Set where

infix 3 ¬_

¬_ : ∀ {ℓ} → Set ℓ → Set ℓ
¬ P = P → ⊥

_≢_ : ∀ {A : Set} → A → A → Set
x ≢ y = ¬ x ≡ y
```

`∀ (a, b : A) : a ≡ b` egy olyan két paraméteres adatszerkezet, aminek akkor van eleme, ha a két paramétere egyenlő, ez az elem pedig a `refl : a ≡ a`, azaz a reflexió axióma szerint definiáltuk. Az ekvivalencia tagadása (≢) az üres halmaz (⊥) bevezetésével valósítható meg, ahol a tagadás az üres halmazra való képzés (`¬ P = P → ⊥`).

Mivel a propozícionális egyenlőség egy ekvivalenciareláció, így a szimmetria (`sym`) és a tranzitivitás (`trans`) is teljesül rá, ezeket szintén definiáljuk.

```
sym : ∀ {A : Set} {a b : A} → (a ≡ b) → (b ≡ a)
sym refl = refl

trans : ∀ {A : Set} {a b c : A} → (a ≡ b) → (b ≡ c) → (a ≡ c)
trans refl refl = refl
```

3.8.2. Tesztelés

A teszteléshez felhasználjuk a nézetből a `step` függvényt, a modellből pedig az állapotot változtató függvényeket, ezek segítségével tetszőleges állapotból eljuthatunk azokba az állapotokba, amikbe a felhasználó is. Példaként tetszőleges állapotból az újrakezdés (`restart`) gombot lenyomva új játék kezdődik. Ennek bizonyítása a definíciókból következik:

```
restart≡newGame : ∀ {s : State} → step s restart ≡ newGame s
restart≡newGame = refl
```

Emellett megadunk egy további függvényt (`»`) a lépések sorozatának jelölésére a könnyebb olvashatóság segítéséhez.

```
_»_ : State → Button → State
state » button = step state button
infixl 10 _»_
```

Ezzel könnyen felírható az első teszteset ($\texttt{test}_1$), ami szerint kezdőállapotból (`init`) 4 szín megadása (`color x`) után az állapotban lévő tipp a megadott 4 szín lesz.

```
test₁ : {n : ℕ} {a b c d : Color} →
  State.guess (init n » color a » color b » color c » color d)
  ≡
  (a :: b :: c :: d :: [])
test₁ = refl
```

A következő tesztesetnél a tippben való léptetésre szeretnénk belátni, hogy nem léphet ki a tippből, azaz a `State.guesspos` értéke nem lehet 4-nél nagyobb. Mivel ennek az értéknek a típusa `Fin 4`, azaz 4 elemű véges halmaz, így erre a típusra kell egy egyszerű lemmát megadnunk. Ez a lemma azt mondja ki, hogy `n` elemű halmaz tetszőleges `i` indexű eleménél `i < n` mindig teljesül.

```
lemma-fin : ∀ {n} → (i : Fin n) → toℕ i < n ≡ true
lemma-fin zero = refl
lemma-fin (suc i) = lemma-fin i
```

Ezután a mi esetünk ennek a lemmának egy speciális esete, ahol `n = 4` és `i = State.guesspos s`, ehhez fel is használjuk az előbb megadott függvényt.

```
test₂ : (s : State) → toℕ (State.guesspos s) < 4 ≡ true
test₂ s = lemma-fin (State.guesspos s)
```

A tesztesetek kiterjedhetnek több gomb működésére is, például beláthatjuk, hogy ha véget ért a játék és nem a 'restart' gombot nyomtuk meg, akkor nem változik az állapot (`step s b ≡ s`) a következő típusú függvény megadásával:

```
no-step-when-ended :
  ∀ {e : End} → (s : State) → (b : Button) →
  b ≢ restart → State.gamestate s ≡ ended e →
  step s b ≡ s
```

Az állításokban hivatkozhatunk az állapot több komponensére is, például szeretnénk belátni, hogy ha egy állapotban (`s`) megváltoztatjuk az aktuális pozíción lévő színt (`modifyColor`), akkor az új állapotban (`modifyColor c s`) az eredeti tipp aktuális pozícióján (`State.guesspos s`) lévő szín a megadott szín lesz:

```
test-modifyColor :
  ∀ {c : Color} (s : State) →
  lookupv (State.guess (modifyColor c s))    -- megváltozott állapot tippjének
    (toℕ (State.guesspos s))                 -- az eredeti állapot tippjének
                                             -- aktuális pozícióján
  ≡ just c                                   -- az új szín lesz
test-modifyColor s = lemma-lookupv (State.guesspos s) (State.guess s)
```

3.8.3. Állítások

Az interfész tesztelése mellett a modell további függvényeire is adhatunk meg állításokat, például szeretnénk belátni, hogy ha a tippünk megegyezik a megoldással, akkor eredményként 4 fekete pöttyöt kapunk, ami megoldása is a játéknak. Ehhez először bebizonyítjuk, hogy a 4 fekete pötty az megoldás, ez mivel egyszerű összehasonlítás, definícióból következik:

```
isSolved-law₁ : isSolved (black :: black :: black :: black :: []) ≡ true
isSolved-law₁ = refl
```

Ezután a `check` függvényről szeretnénk belátni, hogy ha a tipp és a megoldás paramétere megegyezik, akkor eredményül 4 fekete pöttyöt kapunk. Ennek bizonyítása nem triviális, mivel maga a függvény is egy összetettebb algoritmust használ.

Ilyen, bonyolultabb állítások bizonyításához felhasználhatjuk az egyenlőségi érvelést, ami egy speciálisan zárójelezett infix operátor, a tranzitivitást felhasználva több részre bontja a bizonyítást.

```
_≡⟨_⟩_ : ∀ {A : Set} (x : A) {y z : A} → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ refl ⟩ refl = refl

infixr 2 _≡⟨_⟩_

_■ : ∀ {A : Set} (x : A) → x ≡ x
x ■ = refl

infix 2 _■
```

Bizonyítások során használt fogalom továbbá a kongruencia (`cong`), ami azt fejezi ki, hogy ha egy a_1 és a_2 azonos típusbeli értékek megegyeznek, akkor egy rajtuk elvégzett `f` függvény eredménye is megegyezik:

```
cong : ∀ {A : Set} {B : Set} (f : A → B) {a₁ a₂ : A} →
       (a₁ ≡ a₂) → (f a₁ ≡ f a₂)
cong f refl = refl
```

Ennek segítségével a fentebb leírt bizonyításban először belátjuk, hogy a fehér találatok száma 0, így az őket tartalmazó lista üres (`law₁ : whites ≡ [ ]`), ezután levezetjük, hogy a fekete találatokból álló lista 4 fekete elemet tartalmaz (`law₂ : blacks ≡ (black :: black :: black :: black :: [ ])`). Ezeknek az állításoknak a bizonyítására további állításokra és lemmákra van szükség, amiket itt nem részletezünk.

```
check-law₁ : (c : Colors) → check c c ≡ (black :: black :: black :: black :: [])
check-law₁ c =
```

```
  check c c
    ≡⟨ refl ⟩                                  -- check definíciója
  blacks ++ whites
    ≡⟨ cong (λ x → blacks ++ x) law₁ ⟩         -- law₁ felhasználása
  blacks ++ []
    ≡⟨ cong (λ x → x ++ []) law₂ ⟩             -- law₂ felhasználása
  black :: black :: black :: black :: []
    ■                                          -- QED

  where

  law₁ : whites ≡ []
  law₂ : blacks ≡ (black :: black :: black :: black :: [])
```

Ezután ezt a bizonyítást felhasználva egyszerűen megadhatjuk, hogy megegyező tipp és megoldás esetén a játék megoldását kapjuk:

```
check-law₁-solution : ∀ (c : Colors) → isSolved (check c c) ≡ true
check-law₁-solution c =
  isSolved (check c c)
    ≡⟨ cong isSolved (check-law₁ c) ⟩
  isSolved (black :: black :: black :: black :: [])
    ≡⟨ refl ⟩
  true
    ■
```

4. Összegzés

A szakdolgozat elkészítésével sikerült az Agda, egy új és főleg matematikai célokra használt programozási nyelv segítségével egy grafikus felületű, többféle platformon is futtatható programot megvalósítani.

Ennek az implementációnak a legfőbb előnye az, hogy a tesztelést a program kódjának fordításával együtt végzi el a fordító, így futtatható állomány kizárólag a teszteseteknek megfelelő működés esetén jön létre, továbbá nem volt szükség külső tesztelő szoftverek használatára sem, ezzel felgyorsítva a fejlesztés és tesztelés folyamatát.

A funkcionális programok futtatása esetén fellépő hatékonysági problémákat, így például a futás közbeni véletlenszám generálást előre fordított JavaScript függvények használatával sikerült elkerülni, így a mai elvárásoknak megfelelő sebességű program állt elő.

További fejlesztési lehetőségek a programban:

- A Mastermind játék általánosítása, tetszőleges számú tippelési lehetőséggel és tipp hosszúsággal.
- Toplista megvalósítása, azaz I/O műveletek és az agda-frp-js függvénykönyvtár együttes használata, vagy a függvénykönyvtár kiterjesztése

Hivatkozások

[1] agda-frp-js függvénykönyvtár. `https://github.com/agda/agda-frp-js/`, Elérés dátuma: 2013 május 5.

[2] Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, Elérés dátuma: 2013 május 5.

[3] CSS szabvány. `http://www.w3.org/TR/CSS/`, Elérés dátuma: 2013 május 5.

[4] HTML4 szabvány. `http://www.w3.org/TR/html4/`, Elérés dátuma: 2013 május 5.

[5] Andreas Abel. Agda: Equality. `http://www2.tcs.ifi.lmu.de/~abel/Equality.pdf`, Elérés dátuma: 2013 május 5.

[6] ECMA. *ECMA-262: ECMAScript Language Specification.* Third edition, December 1999.

[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[8] A. S. A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2012.

[9] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[10] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing.* Cambridge University Press, New York, NY, USA, 1992.

[11] Unicode Consortium, editor. *The Unicode Standard, Version 6.1 — Core Specification.* 2012.