C++

9. HÉT – KIVÉTELKEZELÉS ÉS MONÁDOK

Kivételkezelés és monádok

-- két egymástól látszólag távol álló téma

Milyen hibák léphetnek fel futás közben?

Általánosságban véve elkerülhetetlen, hogy egy kódot futtatva hibák lépjenek fel

- A fordító (statikus elemzőként) nem tud minden hibát megtalálni (típus-szűkítések, és még nagyon sok minden áteshet a rostán)
- Előfordulhatnak logikai (programozói) hibák, amikre csak futtatáskor derül fény
  - Pl. tömb túlcímzése, vagy egy API nem megfelelő használata (ha szintaktikailag nem megfelelő, az még detektálható, de ha rosszul használjuk fel az eredményt, az kevésbé…)
  - A tömb túlcízmést sok statikus elemző (de nem a fordító) képes detektálni. Ha szemantikai hibát vétünk (hibás feltételezés), az már keményebb dió
- Előfordulhatnak (várt vagy nem várt) futásidejű hibaesemények is
  - Pl. nincs lefoglalható memória, nem érhető el a hálózat, …
  - Ez utóbbiakra elvben felkészülhetünk (erről is szól a kivételkezelés), de konkrétan nem tudjuk, melyik fog melyik futáskor fellépni

Klasszikus megoldás: hibakódok

A klasszikus (nagyon régi) imperatív modellben gyakran dolgoztak hibakódokkal

- A függvények mondjuk egy bool-t vagy intet adtak vissza, amiknek az értéke arra utalt, hogy volt-e hiba.
- Int esetében a hiba fajtájára is lehetett következtetni
- Ilyenkor, ha a függvénynek más visszatérési értéke is lehetne, azt bemenő referenciával vagy pointerrel lehetett kezelni például (output argumentum)

Napjainkban ez a megközelítés ritka, de léteznek alacsonyszintű API-k, ahol még mindig divatos.

A módszer hátulütője, hogy szinte több kód foglalkozik a hiba fennállásának ellenőrzésével, mint magával a programlogikával.

Kivételek (exceptions)

A kivételeket pont azért vezették be sok nyelven, hogy ne kelljen minden függvényhívásnál hibakódot ellenőrizni.

- Különben a kód úgy néz ki, hogy 1 függvény-hívásra szinte szabályként 3-4 sor “overhead” jut

A kivétel felfogható úgy, mint egy objektum, ami:

- Hiba fellépésekor jön létre (mi is létrehozhatunk explicite, vagy egy általunk használt API kódja hozza létre)
- Automatikusan “tovább terjesztődik” a hibakezelés helye felé, ahol végül a kivétel “elkapódik” (vagy ha nincs ilyen hely, akkor crash-el a program)
- A kivételkezelés helye lehet valami egészen más részében a kódnak, mint ahol a hibás függvényhívás történt (vagy ha nincs ilyen hely, akkor crashel a program)

Kivételek (exceptions)

A hibakezelés helye lehet:

- A hibára futó függvény meghívását körbevevő try-catch szerkezet catch blokkja, ha ez a catch blokk alkalmas az adott kivételtípus elkapására
- A hibára futó függvény hívását tartalmazó F függvényt meghívó F1 függvényben levő try-catch szerkezet catch blokkja, ha a szerkezet try blokkjában benne van az F függvény meghívása és a catch blokk alkalmas az adott kivételtípus elkapására
- Az F1 függvényt meghívó F2 függvényben levő try-catch szerkezet catch blokkja, ha a szerkezet try blokkjában benne van az F1 függvény meghívása és a catch blokk alkalmas az adott kivételtípus elkapására, és így tovább

Mivel ilyenkor a futtatási környezetnek vissza kell mennie a függvényhívási láncolatban egészen addig a függvényig, ahol a programozó definiált a kivétel “elkapására” alkalmas catch blokkot, ezt a működést úgy is hívják, hogy “stack unwinding”

- A stacken levő hívás-láncolat “visszatekeredik”, mint egy cérnagömbölyeg

Kivételek – előnyök és hátrányok

A kivételek legfőbb **előnyei**:

- Az API-t meghívó kódot (programozót) rákényszerítik, hogy gondolkozzon el, helyesen használja-e az API-t és hogy milyen váratlan esetek léphetnek fel
- Az API-t készítő kódot (programozót) rákényszeríti, hogy tegye világossá, milyen feltételekkel biztosítja működését (milyen kvázi “vis maior” esetek fellépésekor nem vállalja a megfelelő működést és dob kivételt)
  - Ha az API olyan kivételt dob, ami nincs előre lefektetve a dokumentációban, az API kódja hibás
  - Ha az API olyan kivételt dob, ami előre le lett fektetve a dokumentációban, de a kivétel nincs elkapva sehol, akkor az API-t hívó kód hibás
  - -> éppen ezért szokás az API-knak saját kivétel típusokat definiálni, hogy ne legyen értelmezési kavarodás
- Mivel a stack unwinding mechanizmus automatikus, nem kell a függényhívási láncolat különböző szintjein “gondolkozni” – a kivételek a megfelelő helyen elkaphatóak. Magyarán a kivétel fellépésének és kezelésének a helye jól elszeparált
- A kivétel fellépése és kezelése közötti kódban létrejött változók felszabadítása előre definiált szabályok szerint történik

Kivételek – előnyök és hátrányok

A kivételek legfőbb **hátrányai** (Legalábbis a funkcionális programozás “felkent papjai” szerint):

- Ha egy függvény kivételt dob, már nem lehet tiszta (pure), mivel ugyanarra a bemenetre két különböző választ is adhat
  - ha az egyik esetben mondjuk nem érhető el a hálózat vagy elfogy a memória, míg a másikban nem
- Mi történik, ha tolerálni szeretnénk, hogy egy függvény egy esetben hibára futott, de 99 másikban nem futott hibára?
  - Pl. ha egy konténer elemein futtatjuk egyazon függvényt (map), de az egyik hívás sikertelen, ez még nem jelenti azt, hogy a többi eredményt el kell, hogy dobjuk
- Majd látni fogjuk, hogy a funkcionális programozási paradigma sokkal megengedőbb a hibákkal szemben (persze azokat ugyanúgy kezelni kell, de nem szeretik megakasztani csak ezért a teljes “információ-feldolgozási láncot” – többek között ezért is dolgoztak ki olyan szerkezeteket, mint a monadikus típusokat, melyek a hibák és egyéb mellékhatások fellépését mintegy objektumként modellezik)

Kivételek dobása C++-ban

A kivétel a C++-ban egy ugyanolyan objektum, mint bármelyik másik

Ideális esetben az objektum típusa származik egy kivétel ősosztályból, de ez nem követelmény. Egy stringet, vagy intet is használhatunk kivételként, ha a ***throw*** utasítással “eldobjuk”

Ehhez mindössze annyi szükséges, hogy a ***throw*** kulcsszóval a hiba fellépésekor eldobjuk a kivételt

- A kivételt elkapni egy try-catch blokkal lehetséges, feltéve, hogy a kivétel egy olyan kódrészből kerül eldobásra, amely a try blokk része
- A catch blokkban megadható, hogy milyen típusú kivételekre vagyunk receptívek. Több catch blokk is készíthető egyszerre, ld. következő fólián levő példa

```cpp
#include <iostream>

void f(int input) {
    if (input == 1) {
        throw "String Exception";
    }
    if (input == 2) {
        throw 55;
    }
    std::cout << "No exception was thrown" << std::endl;
}

int main() {
    try {
        f(1);
        f(2);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    }
    return 0;
}
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
terminate called after throwing an instance of 'char const*'
exited, aborted
>
```

Példa: throw, try, catch

Itt a kivételt nem sikerült elkapnunk, mert a catch blokkban csak int-ekre figyelünk:

```cpp
#include <iostream>
#include <string>

void f(int input) {
    if (input == 1) {
        throw "String Exception";
    }
    if (input == 2) {
        throw 55;
    }
    std::cout << "No exception was thrown" << std::endl;
}

int main() {
    try {
        f(1);
        f(2);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    } catch (const char* e) {
        std::cout << e << std::endl;
    }
    return 0;
}
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
String Exception
>
```

Példa: throw, try, catch

A kivételt ebben az esetben csak const char* típusúként lehet elkapni – mivel automatikusan nincs kasztolás

Dobás érték szerint, elkapás referenciával

A throw kulcsszó bármilyen objektumot alapvetően érték szerint továbbít. Érdemes ezeket a catch blokkban referenciaként elkapni (hacsak nem pointerről van szó).

Ennek az az előnye, hogy futáskor a fedél alatt nem kell lemásolni az objektumot, + dinamikus polimorfizmus is használható

- Ha egy konkrét típusú kivételt dobunk, elkapható annak egy szülőosztályára mint típusra hivatkozó referenciával is
- Ha a kivétel értékét nem módosítjuk, elkaphatjuk const referenciaként is

std::exception

Általában azonban nem bármilyen, hanem valamilyen, az std::exception osztályból származtatott típust célszerű használnunk. A sztenderd könyvtár által dobott kivételek mind ilyenek:

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)

- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
  - tx_exception(TM TS)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - format_error(C++20)

- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- bad_variant_access(C++17)

std::exception

Az std::exception osztály tartalmazza az alábbi virtuális metódust:

```
virtual const char* what() const noexcept;
```

Ennek a felüldefiniálásával tudjuk elérni, amikor kiterjesztjük ezt az osztályt, hogy a felhasználó a catch blokkban információt kérjen az elkapott hibáról

noexcept

A C++11 óta a *noexcept* kulcsszóval jellemzett függvények / metódusok esetén a fordító kikényszeríti, hogy az adott függvény / metódus ne dobjon kivételt. Interfészek deklarálásánál ez hasznos információ tud lenni más programozók számára, ahogy pl. a move konstruktor és assignment esetén szinte kötelező is használni, mert a fordító akkor tudni fogja, hogy biztonságosan mozgathat.

```cpp
#include <iostream>
#include <string>

void f(int input) noexcept {
    if (input == 1) {
        throw "String Exception";
    }
    if (input == 2) {
        throw 55;
    }
    std::cout << "No exception was thrown" << std::endl;
}

int main() {
    try {
        f(1);
        f(2);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    } catch (const char* e) {
        std::cout << e << std::endl;
    }
    return 0;
}
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:9:9: warning: 'f' has a non-throwing exception
      specification but can still throw [-Wexceptions]
        throw 55;
        ^
main.cpp:4:6: note: function declared non-throwing here
void f(int input) noexcept {
     ^           ~~~~~~~~
main.cpp:6:9: warning: 'f' has a non-throwing exception
      specification but can still throw [-Wexceptions]
        throw "String Exception";
        ^
main.cpp:4:6: note: function declared non-throwing here
void f(int input) noexcept {
     ^           ~~~~~~~~
2 warnings generated.
> ./main
terminate called after throwing an instance of 'char const*'
exited, aborted
>
```

Monoidok, functorok és monádok

Kategóriaelméleti típusok, melyek többek között a kivételkezelésre is alternatív megközelítést nyújtank

Monoidok, functorok és monádok

A címben szereplő fogalmak kategóriaelméleti fogalmak.

A kategóriaelmélet a múlt század közepe táján alakult ki abból a célból, hogy látszólag különböző, de szerkezetükben azonos problémákat matematikailag egységesen lehessen kezelni.

- Ha egy halmazról és az elemein elvégezhető műveletről tudom, hogy egy adott kategóriába tartozik, érteni fogom, hogy nagyjából milyen tulajdonságokat várhatok el tőle, és (esetünkben) hogyan tudom hatékonyan leprogramozni
- Esetünkben túlzottan mélyen nem megyünk bele ennek a matematikájába – nem szükséges ahhoz, hogy az alapelveket megértsük és használni tudjuk!

Monoidok

Monoidnak nevezünk minden olyan T típusból és fölötte értelmezett, bináris op operátorból álló együttest, ahol:

- op asszociatív, tehát T típusú t1, t2 és t3 esetén **(t1 op t2) op t3** ugyanaz, mint **t1 op (t2 op t3)**
- létezik olyan speciális T típusú **e** érték (“üres”, vagy “null elem”), amire igaz, hogy bármely T típusú t-re: **e op t = t**

Ez tehát azt jelenti, hogy monoidoknál teljesen mindegy, hogy egy műveletláncolatban melyik műveletet végezzük el először, és melyiket másodszor – csak az operandusok sorrendje számít (op asszociatív, de nem kommutatív). Ez fontos!

Példák monoidokra:

- unsigned int típus std::max operatárorral (ekkor e = 0, mivel 0 és bármilyen t maximuma t lesz – feltéve persze hogy t pozitív – a maximálás mint művelet pedig nyilvánvalóan asszociatív)
- std::optional<X> típus is tud monoid lenni egy jól meghatározott append művelettel, ha X valamilyen monoidnak a T típusa. Ugyanis (ld. következő fólia)

std::optional mint monoid

```
template <class SomeMonoidType>
std::optional<SomeMonoidType> append(
      std::optional<SomeMonoidType> lhs,
      std::optional<SomeMonoidType> rhs,
      SomeMonoidType (*operatorOfSomeMonoidType)(SomeMonoidType, SomeMonoidType)) {
  if (!lhs) return rhs;
  if (!rhs) return lhs;
  return std::optional<SomeMonoidType>(operatorOfSomeMonoidType(*lhs, *rhs));
}
```

std::optional mint monoid

Az előző fólián a monoid típusa std::optional<SomeMonoidType>, operátora pedig **append**. A null elem az std::nullopt, amivel ha ezt az értéket bármihez appendoljuk, a másik operandust kapjuk vissza. Alapesetben, ha egyik operandus sem std::nullopt, a SomeMonoidType operátorát használjuk az appenden belül is.

SomeMonoidType pedig a korábbiak alapján lehet pl. **unsigned int**, úgy, hogy a hozzá tartozó operátor pedig **std::max**. Appendet ez esetben így használhatjuk fel:

```
std::optional<unsigned int> a{ 6u };

std::optional<unsigned int> b{ 8u };

std::optional<unsigned int> c{std::nullopt};

auto unsignedIntOp = [](unsigned int a, unsigned int b) { return std::max(a, b); };

std::optional<unsigned int> result = append<unsigned int>(

  append<unsigned int>(a, b, unsignedIntOp),

c, unsignedIntOp);

std::cout << "maximum of 6, 8 and nullopt is: " << *result << std::endl;
```

std::optional mint monoid

Az előző fólián látható kód kinézetre nem annyira szép, elsősorban az operator prefix operátorként való alkalmazása miatt – melyet az alábbi példa szemléltet:

```
Op(Op(elem1, Op(elem2, elem3, fuggveny), fuggveny), elem4, fuggveny)
```

Nyilván felesleges, hogy újra meg újra meg kelljen adnunk ugyanazt a függvény nevet (a példában unsignedIntOp-ot), plusz olvashatóbb lenne az egész, ha Op infix operator lenne:

```
elem1 Op elem2 Op elem3 Op elem4
```

F: A->A függvények kompozícióval, mint monoidok

Ettől függetlenül az egész monoid témának a jelentősége óriási, hiszen ha egy típus-művelet párról tudjuk, hogy monoid, akkor sok azonos típuson bármilyen sorrendben végrehajthatjuk az adott műveletet, akár párhuzamosítva is – egyedül az operandusok sorrendjét kell tiszteletben tartani. Ráadásul a kódunk is sokkal elegánsabb lesz.

Vegyünk egy példát. Hasznos tudni, hogy az egybemenetű és a bemenetük típusával azonos típust visszaadó függvények (jelöljük ezeket úgy, hogy f: A -> A) monoidok, ha operátorként a kompozíciót tekintjük. A null elem ebben az esetben az identitás függvény, amely a bemenetét egy-az-egyben visszaadja.

Ez azt jelenti, hogy:

$$f\ (g\ (h\ (x))) = fg\ (h\ (x)) = f\ (gh(x))$$

feltéve persze, hogy f, g és h az x típusának megfelelő bemenetet vár, és ugyanilyen típusú kimenetet ad. Mindez – egy kicsit belegondolva – könnyen belátható.

Függvénykompozíció - példa

Az alábbi kód például csak úgy kicsattan az eleganciától ☺

```
template <class A> using AtoAFunc = std::function<A(A)>;

template <class A> class FnCompMonoid {

    AtoAFunc<A> func;

public:

    FnCompMonoid(AtoAFunc<A>& f) : func{ f } {}

    A operator()(A input) const { return func(input); }

    static FnCompMonoid<A> append(FnCompMonoid<A>& left, FnCompMonoid<A>& right) {

        AtoAFunc<A> composed = [left, right](A input) -> A { return left(right(input)); };

        return FnCompMonoid<A>(composed);

    }

};
```

```
int main() {

    AtoAFunc<int> times2 = [](int x) {return 2 * x; };

    AtoAFunc<int> plus2 = [](int x) {return 2 + x; };

    AtoAFunc<int> ident = [](int x) {return x; };

    std::vector<FnCompMonoid<int>> vec = { times2, plus2 };

    auto aggregatedMonoidFunc = std::accumulate(

        vec.begin(), vec.end(),

        FnCompMonoid<int>(ident), FnCompMonoid<int>::append

    );

    std::cout << "aggregatedMonoidFunc(" << 5 << ") = ";

    std::cout << aggregatedMonoidFunc(5) << "\n";

}
```

Functorok

A functoroknak is van szigorúan vett matematikai (kategóriaelméleti) definíciója. Itt most programozás-szemléletű definíciót adunk anélkül, hogy a matematikába mélyebben belemennénk.

- Vigyázat! A functor szót itt kategóriaelméleti értelemben használjuk, nem úgy, mint korábban (a korábbi functort mostantól függvény objektumnak nevezzük).

Functornak nevezünk minden olyan template-elt osztályt, melynek 1 template típusa van (tehát mint Functor<T>), és amelyre definiálva van egy map nevű meghívható egyed (callable) az alábbi tulajdonságokkal:

- map(f, a) kifejezés a.cs.a. legális kifejezés, ha:
  - a egy Functor<T> típus
  - f egy meghívható egyed (callable), ami T típusú bemenetet vár és U típusú kimenetet ad
  - map(f, a) kimenetének típusa Functor<U> lesz
- Ha f(x) = x minden x bemenetre, akkor map(f,a) eredménye pontosan a lesz
- map(g, map(f,a)) pontosan ugyanazt jelenti, mint map(gf, a), ahol:

```
auto gf = [f, g](auto t) { return g(f(t));}
```

Functorok

Az utolsó két kitételen nem kell sokat rugózni, lényegében azt jelentik, hogy a functorok úgy viselkednek, ahogy várnánk, illetve hogy a functoroknak is valamilyen kapcsolatuk van a függvény-kompozícióval.

Intuitív értelmezés: a functorok olyanok, mint a konténerek, amiknek a belsején műveleteket lehet elvégezni. Van bennük ugyanis valamilyen T típussal jellemezhető valami, és a map ezen a valamin tud operálni – függetlenül attól, hogy a konténer pontosan milyen típus.

- A jól ismert std::vector<T> vagy std::list<T> típusok functorok (lehetnek, ha definiálunk hozzájuk map függvényt)! Egy adott vektorban vagy list-ben T típusú egyedek vannak, a map pedig végigmegy mindegyiken és végrehajt rajtuk egy adott műveletet. Végül az eredmény std::vector<U> vagy std::list<U> lesz. És ez működik, függetlenül a konténer típusától!

Functorok

Emellett absztraktabb szinten bármi lehet functor – az is, amire nem konténerként gondolunk.

- A lényeg, hogy a map függvény kicsomagol valamit, a beltartalmára végrehajtja az adott műveletet, majd az eredményt visszacsomagolja

Functornak számít pl. az std::optional típus is. Ugyanis a típusnak van egy “belseje”, ami – ha létezik – operandusként szolgálhat a mappelt műveletnél. Vagyis:

```
template <typename T, typename U>

std::optional<U> map(const std::function<U(T)> &f, const std::optional<T>& a) {

  if (a) {

    return std::optional<U>(f(*a));

  }

  return std::nullopt;

}
```

Functorok

Vagy vegyünk egy összetettebb példát. Legyen egy std::vector-unk, amely std::optional<int>-eket tartalmaz. Tegyük fel, hogy a célunk az, hogy mindegyik int helyett stringünk legyen.

Ez egy konténeren belüli konténer elemeire végrehajtandó map.

```
std::vector<std::optional<std::string>> getStrings(const std::vector<std::optional<int>>& ints) {
    std::function<std::string(int)> f1 = [](auto i) {return std::to_string(i); };
    std::function<std::optional<std::string>(std::optional<int>)> f2 = [&f1](auto o) {return map(f1, o); };
    auto f3 = [&f2](auto v) {return map(f2, v); }; // from vector of optional T to vector of optional U
    return f3(ints);
}
```

Functorok

Miben rejlik a functorok jelentősége?

- A map, mint művelet kényelme egy dolog. De ha madártávlatból nézzük a functorokat, és a monoidokkal összevetjük őket, kiderül, hogy itt is beszélhetünk egyfajta függvény-kompozíció könnyebbé tételéről!
- Emlékezzünk vissza, hogy a monoidok az F: A -> A típusú függvények kompozícióját tette lehetővé.
- A functorok ezzel szemben az F: A -> W(A) típusú függvények kompozícióját teszi lehetővé, ahol W egy wrapper
  - A map() függvényt, mint operátort felfogva, egy olyan operátort kapunk, amelyik az ilyen függvények komponálását könnyíti meg.

Nézzünk minderre egy a monoidos példához hasonlóan elegáns bemutató példát!

Functorok

Az alábbi kódban hasonlót látunk, mint a monoidos példa esetében, csak itt az összekomponált függvények minden esetben std::optional-ba csomagolják az eredményüket.

```
template <class A, template<class> class W> using AtoWAFunc = std::function<W<A>(A)>;

template <template<class> class W, typename A> class FnCompFunctor {

    AtoWAFunc<A,W> func;

public:

    FnCompFunctor(AtoWAFunc<A,W>& f) : func{ f } {}

    W<A> operator()(A input) const { return func(input); }

    static FnCompFunctor<W,A> append(FnCompFunctor<W,A>& left, FnCompFunctor<W,A>& right) {

        AtoWAFunc<A,W> composed = [left, right](A input) -> W<A> { return left(*right(input)); };

        return FnCompFunctor<W,A>(composed);

    }

};
```

```
int main() {

    AtoWAFunc<int, std::optional> ftimes2 = [](int x) {return std::optional<int>(2 * x); };

    AtoWAFunc<int, std::optional> fplus2 = [](int x) {return std::optional<int>(2 + x); };

    AtoWAFunc<int, std::optional> fident = [](int x) {return std::optional<int>(x); };

    std::vector<FnCompFunctor<std::optional, int>> vec = { ftimes2, fplus2 };

    auto aggregatedFunctorFunc = std::accumulate(vec.begin(), vec.end(),

        FnCompFunctor<std::optional, int>(fident), FnCompFunctor<std::optional, int>::append

    );

    std::cout << "aggregatedFunctorFunc(" << 5 << ") = ";

    std::cout << *aggregatedFunctorFunc(5) << "\n";

}
```

Functorok

Ezzel a módszerrel el lehet érni például, hogy egy hosszú művelet-láncban ne okozzon problémát, ha valamelyik köztes művelet sikertelen

- Ehhez annyi kell, hogy az összekomponált függvények kezelni tudják, ha a bemenetük valójában std::nullopt
- Ezt az “intelligenciát” beletehetjük a map() függvénybe

Tágabb értelemben a functorok hasznosak, ha egy vagy több alap-függvény viselkedését ki akarjuk egészíteni

- A map() függvény megteheti, hogy a functor beltartalmát kicsomagolja, végrehajt valami műveletet rajta, az eredményel valami mást is csinál – pl. logolja egy fájlba – majd az eredményt visszacsomagolja
- Ilyen értelemben, ha van egy X típusunk valamilyen op művelettel, akkor létrehozhatunk egy tetszőleges F<X> típust, melynek map függvénye ugyanazt csinálja, mint op – de mellette járulékos műveleteket is elvégez.

Monádok

Most pedig térjünk rá a monádokra.

- Az Internet tele van monad tutorialokkal, mindegyik azt ígéri magáról, hogy a többivel ellentében ő lesz az, amelyik végre elmagyarázza, mik is azok a monádok.

Ennek talán az is az oka, hogy sok szemszögből lehet nézni, hogy mik azok a monádok.

Kezdjük ott, hogy a funkcionális programozás egyik alapvetése, hogy a mellékhatás az rossz dolog

- Mit értünk mellékhatás alatt?
- Azt már tudjuk, hogy a globális változók használatát érdemes kerülni, mert az egész programból elérhetőek és belegárgyulunk, ha meg akarjuk keresni, hogy hány helyen állítottuk az adott értéket. Tehát egy olyan függvény, amelyik globális változót ír, annak működése egyfajta mellékhatással jár.

Monádok

- Általában véve minden olyan viselkedését egy függvénynek mellékhatásként tartjuk számon, ami azt eredményezi, hogy valamelyik (akár maga, akár egy másik) függvény 2x ugyanúgy meghívva nem ugyanazt az eredményt fogja adni.
  - Ha meghívom azt a függvényt, hogy x a négyzeten, a 4 bemenetre, mindig 16-ot kell, hogy kapjak. Ez a tesztelhetőség alapja.
  - Ezért ezeket a fajta függvényeket szeretjük
  - De ha van egy függvényem, aminek 2x ugyanazt a bemenetet adom de mást ad vissza, mert egy globális változó értéke a két esetben más, az nehezen karbantartható programhoz vezet
  - A funkcionális programozásban ezt nagyon komolyan veszik, és még a kivételeket is mellékhatásként fogják fel – mert a függvény futása megszakad és “valahol máshol” a programban folytatódik a futás. Ez akkor rossz, ha mondjuk 2x ugyanazt a bemenetet adom, de a 2. esetben mondjuk kivétel lép fel és ezért megszakad a normál futás.

Monádok

No, ez mind szép és jó. Próbáltak is olyan nyelveket készíteni, amik tisztán funkcionálisak (pl. Haskell). De mi történik, ha egy program interaktálni akar a külvilággal – user bemenet? kimenet a konzolra? olvasás egy adatbázisból?

- Ezek mind mellékhatások! Ha egy adatbázisból más értéket olvasok, hiába adtam ugyanazt a bemenetet a függvénynek
- És mi történik, ha megszakad a hálózati kapcsolat, és nem tudok az adatbázisból olvasni? Etc. etc.

Ezeknek a problémáknak a kezelésére találták fel a monádokat. A monádok tulajdonképpen egy komputációt reprezentáló típusként foghatóak fel

- “Igen, lehet, hogy a komputáció mellékhatással tud járni, de ezeknek a lehetséges mellékhatásoknak az összességét legalább egy típusban tudjuk reprezentálni és így az eshetőségekkel tudunk számolni.”
- Jó példa erre az std::optional. Ez a típus bármikor felveheti az std::nullopt értéket – ezzel reprezentálni tudjuk, hogy a számítás valamilyen okból sikertelen. De ha sikeres, akkor ki tudjuk belőle olvasni az eredményt.
- No de akkor nézzük, mikor lesz egy típus monad!

Monádok – fél-formális definíció

Funkcionális programozásban az M<A> típusú változó monad, ha az egy functor is egyben, melyre a jól ismert map(f,a) függvényen kívül további két függvény definiálva van:

- pure(a) – vagy más néven unit(a), amely egy A típusú a-ból M<A>-t csinál (ez olyan, mint egy konstruktor)
- join(M<M<A>>), amely egy M<M<A>> típusból csinál M<A> típust – magyarán kicsomagolja a monad belsejét, ha az is egy monád
  - Az egyetlen formális feltétel erre vonatkozóan, hogy ha többszintű beágyazás van, akkor a joinolás sorrendje mindegy.
- Sok esetben a map-nek és join-nak együttesen külön nevet adnak: flatmap(), vagy bind()

Ha jól belegondolunk, a join() / flatmap() függvénynek az lesz az értelme, hogy olyan függvényeket is lehessen map()-elni, melyek A típusból M<B> típust csinálnak. A map()-nek ilyenkor egy M<M<B>> típus lenne a kimenete (mivel a map() szabály szerint az eredményt visszacsomagolja a wrapper típusba – jelen esetben M-be). join() vagy flatmap() segítségével ezt a struktúrát “ki lehet lapítani” (flatten)

Monádok – mappelés szteroidokon

Tegyük fel például, hogy van egy vektorom intekkel. Az most a célom, hogy mindegyik intet háromszor ismételjek meg (egy 3x olyan hosszú vektort szeretnék).

Ezt megoldhatom hagyományosan úgy, hogy végig iterálok rajta, kiolvasok minden elemet és háromszor belenyomom egy másik vektorba.

A funkcionális megoldás az lenne, hogy rá mappelek egy függvényt, ami 3x ugyanazt az elemet visszaadja… de mibe csomagolva? egy vektorba?

- Ha így csinálom, akkor egy vektorokból álló vektort kapok!
- És itt lesz hasznos a join, ami végül egy vektorokból álló vektor helyett egy sima vektort fog nekem visszaadni.

Magyarán: a join egy olyan magasszintű fogalom, amelynek segítségével nagyon tömören le lehet írni olyan műveleteket, amik ilyen rejtett be- és kicsomolásokat használnak.

Monádok – egy módszer mellékhatások formalizációjára

Na de mi köze ennek az egésznek a mellékhatásokhoz?

- Hát az, hogy – ha visszaemlékszünk, azt mondtuk – egy monad az egy számítás enkapszulálása is lehet egy típusban

Tehát M<A> flatmap() függvénye csinálhatja egyrészt azt, hogy kiírja a konzolra a benne levő A típust, majd végrehajt rajta egy műveletet és az eredményt, M-be becsomagolva visszaadja.

- Ez önmagában nem új gondolat, hiszen a functor map() függvényéről is elmondtuk ugyanezt

Ezen túlmenően a monad viszont alkalmas lesz arra is, hogy a kiírás tényét, vagy annak tartalmát egy összetettebb adatstuktúrában reprezentálja is!

- Mindez úgy lehetséges, hogy olyan f: A -> M<B> függvényeket használunk, melyek A típusból B-t is csinálnak, de M-ben enkapszulálják a kiírás tényét vagy tartalmát is (a külvilági hatás így a program állapotában benne van)
- flatmap(), vagy más néven bind() pedig annyit csinál, hogy egy már létező M<B>-nek a B-jére alkalmaz egy ilyen függvényt, majd a külvilági hatásokat az eredményül kiadott M-ben összekonkatenálja.
- Magyarán: a monad az nemcsak egy járulékos viselkedést tesz lehetővé, mint a functor, hanem annak a lenyomatát is magában hordozza.

Brian Beckman, aki egy nagy tudós (dolgozott a NASA-nál, Microsoftnál) azt mondta, hogy egy monad egy rakétát is elindíthat. De ami a nyelv szempontjából a lényeg, hogy ennek a műveletnek most már típusa van, amit egy fordítóprogram ellenőrizni tud, és a programozó is érti, hogy ennek a típusnak ez egy velejárója. Ha nem ezt szeretné, akkor egy másik függvényt kell meghívnia, ami nem M<A> típust ad vissza!

Monádok – példa: Writer monad

Erre mutat példát az alábbi Writer monad:

```
template <typename A> class WriterMonad {
    std::vector<std::string> logs; A result;
public:
    WriterMonad(A a, const std::vector<std::string>& v) : result{ a }, logs{v} { }
    static WriterMonad<A> unit(A a) { return WriterMonad<A>(a, std::vector<std::string>()); }
    WriterMonad<A>& bind(std::function<WriterMonad<A>(A)> func) { // a.k.a. flatmap()
        WriterMonad<A> temp = func(result);
        result = temp.result;
        logs.push_back(temp.logs.at(0));
        return *this;
    }
```

Monádok – példa: Writer monad

A print() és main() függvények pedig így néznek ki:

```
    void print() {
        std::cout << "value is: " << result << ". order of operations was: " <<
           std::endl;
        for (std::string& s : logs) { std::cout << s << ", "; } std::cout << std::endl;
    }
};
```

```
value is: 25. order of operations was:
added one, multiplied by two, multiplied by two, added one,
```

```
int main()
{
    std::function<WriterMonad<int>(int)> addOne = [](int x) -> WriterMonad<int> {
        return WriterMonad<int>(x + 1, std::vector<std::string>{"added one"});
    };
    std::function<WriterMonad<int>(int)> multTwo = [](int x) -> WriterMonad<int> {
        return WriterMonad<int>(x * 2, std::vector<std::string>{"multiplied by two"});
    };
    WriterMonad<int> x = WriterMonad<int>::unit(5);
    x.bind(addOne).bind(multTwo).bind(multTwo).bind(addOne);
    x.print();
}
```

Monádok – kapcsolódás a hibakezeléshez

A példa alapján az is jobban körvonalazódhat a fejünkben, hogy a monádok hogyan segíthetnek a hibakezelésben.

Ha van egy hosszú számítási láncolatunk, ami bárhol “elhasalhat” egy hiba miatt, egy Failure monad használata mellett nem kell minden lépésben ellenőriznünk, hogy történt-e hiba

- De ami lényeges, hogy nem kell a futást megszakítanunk sem!

A megoldás lényege, hogy a hiba tényét a monadon belül dokumentáljuk. Később, miután az egész láncolat végig fut, a dokumentált hiba alapján – a változók belső állapotából, ami a debuggerben is látszik! – ragyogóan meg lehet érteni, hogy pontosan hol, melyik ponton lépett fel a hiba.