IB015 – Domácí úkol 12: Implementace procesoru

Termín: do 11. 1. 23.59; způsob odevzdání je popsán níže.

V posledním domácím úkolu si implementujeme vlastní procesor. Rok 2020 byl z tohoto hlediska plný zajímavých počinů[1], tak proč se nepřidat?

Oproti stávajícím „standardním“ procesorům bude ten náš doslova drobek. Disponuje velmi zjednodušenou instrukční sadou, třemi registry a pamětí. Samotný procesor již dostanete definovaný, vaším úkolem bude implementovat několik funkcí, které procesor emulují a vykonávají jeho instrukce.

Reprezentace procesoru

Jak bylo řečeno, procesor disponuje jedinou pamětí pro data, která je reprezentována datovým typem `Memory` – pro celou definici a popis práce s ní, vizte příslušnou kapitolu. Dále máme k dispozici sadu tří registrů reprezentovanou datovým typem `Registers`.

```
type Value = Integer

data RegisterName = R1 | R2 | R3 deriving (Show, Eq)
data Registers = Registers Value Value Value deriving (Show, Eq)
```

Mimo paměť má každý procesor i svou instrukční sadu, v našem případě reprezentovanou datovým typem `Instruction`. Instrukce a práce s nimi, respektive efekty jednotlivých instrukcí jsou popsány v této části. U instrukcí, které pracují s adresami, rozlišujeme adresy relativní a absolutní datovým typem `AddressType`.

```
data AddressType = Relative | Absolute deriving (Show, Eq)
```

Dále definujeme několik dalších datových typů. Některé instrukce jsou schopné produkovat výstup. Pro ten máme dedikovaný typ `Output`.

```
data Output = S String | N Value deriving (Show, Eq)
```

Program potom reprezentujeme stejně pojmenovaným typem `Program`. Jde ve skutečnosti o typový alias pro paměť obsahující instrukce. Ačkoliv paměť s instrukcemi nám stačí pouze číst a není potřeba ji modifikovat, používáme stejný datový typ jako pro data – usnadní nám to práci a zároveň nám to umožní využívat stejné (nejen) pomocné funkce jako pro data.

Pro samotná data, se kterými pracujeme, máme datový typ `Data`. I toto je pouhý typový alias pro paměť hodnot. Hodnoty jsou pro nás pouze celá čísla – `Integer`, ale pro lepší přehlednost a případnou rozšiřitelnost si definujeme odpovídající typový alias.

```
type Program = Memory Instruction
type Data = Memory Value
```

Práce s pamětí

Paměť v našem procesoru si můžete představit jako dlouhou pásku, která je rozdělená na jednotlivá políčka. Z těchto políček můžeme nejen číst hodnoty, ale můžeme nové hodnoty i zapisovat. Mezi políčky se zároveň můžeme přesouvat. Vždy ale můžeme číst nebo zapisovat pouze do jednoho políčka – tomu říkáme aktuální/zaměřené políčko, které je jednoznačně určené indexem v rámci paměti (vizte níže).

Paměť reprezentujeme datovým typem `Memory a`, jehož definice je následující:

```
data Memory a = Memory Integer [a] [a] deriving Show
```

Paměť se tedy skládá z čísla a dvou seznamů. Číslo značí stávající adresu/index (pozici) v rámci paměti. Seznamy pak odpovídají jejímu obsahu, přičemž první seznam reprezentuje obsah paměti **před** aktuální adresou (indexem), druhý seznam potom reprezentuje obsah paměti **na** indexu (hlavička seznamu) a paměť **po** indexu (zbytek – ocásek – seznamu).

[1] Intel přišel se svou hybridní architekturou Lakefield, AMD vydalo vynikající Zen 3 a Apple začal využívat své vlastní ARMové řešení i v počítačích.

```
Memory 0 [] [1, 2, 3, 4] -- výchozí pozice
Memory 1 [1] [2, 3, 4]   -- posun o jedna doprava
Memory 2 [2, 1] [3, 4]   -- další posun o jedna doprava
```

Posuny v paměti způsobují nejen změnu indexu, ale i přelévání prvků mezi seznamy. Posun doleva způsobuje snižování indexu a přesun prvků z prvního seznamu na začátek druhého seznamu. Posouvání doprava index zvyšuje a prvky ze začátku druhého seznamu vkládá na začátek prvního.

Všimněte si, že první seznam je obsahově obrácený. Důvodem je, že seznamy v Haskellu jsou jednostranně zřetězené, a je tedy výhodnější si obsah paměti před indexem pamatovat v opačném pořadí tak, abychom se mohli efektivně pohybovat pamětí nejen dopředu, ale i dozadu (vždycky nám stačí vzít si jeho hlavičku, což nevyžaduje průchod seznamem).

V případě, že má dojít k posunu, ale ten není možný (typicky hledaný index neexistuje), zavolejte funkci `error` s pokud možno smysluplnou chybovou hláškou.

Nápověda: Není od věci zavést si pomocné funkce pro práci s pamětí (typicky funkce, které *nějak* pracují s obsahem aktuálního políčka atp.), které vám pomohou zpřehlednit kód a omezit jeho duplikaci.

Funkce k implementaci

Vaším úkolem je naprogramovat následující sadu funkcí.

- `fromList :: [a] -> Memory a`

  Tato funkce dostane na vstupu seznam hodnot, ze kterých vytvoří paměť. Výsledná paměť je zaměřená na začátek vstupního seznamu – to znamená, že index je 0 – a je omezena velikostí vstupního seznamu, takže validní adresy (políčka) a jejich hodnoty jsou dány vstupním seznamem (viz práce s pamětí).

  ```
  fromList [1, 2, 3]     ~>* Memory 0 [] [1, 2, 3]
  fromList [True, False] ~>* Memory 0 [] [True, False]
  ```

  Protože výstupem z této funkce je vždycky paměť, která je alespoň v jednom směru konečná, nazýváme takovou paměť *pamětí omezenou*.

- `fromListInf :: a -> [a] -> Memory a`

  Tato funkce dostane výchozí hodnotu a seznam hodnot, ze kterých vytvoří paměť. Na rozdíl od předchozí funkce není tato paměť nijak omezená v žádném směru – políčka mimo vstupní seznam obsahují výchozí hodnotu a přípustné jsou i záporné indexy. To znamená, že neexistuje adresa (index), na kterou nelze přistoupit. Počáteční index je opět roven nule a paměť je zaměřená na hlavičku vstupního seznamu.

  Paměť, kterou dostaneme po aplikaci této funkce, nazýváme *nekonečnou*, či *neomezenou*.

- `focusRel :: Integer -> Memory a -> Memory a`

  Tato funkce bere jako první parametr hodnotu, o kterou se chceme posunout vůči stávající pozici, a paměť, na které chceme posun provést.

  Výstupem je paměť, která je zaměřená na novou pozici. Ke změně zaměření je potřeba nejen správně změnit index aktuálně zaměřené části paměti, ale je třeba vhodně modifikovat i oba seznamy.

  ```
  focusRel 2 (Memory 0 [] [1, 2, 3, 4])  ~>* Memory 2 [2, 1] [3, 4]
  focusRel 1 (Memory 2 [2, 1] [3, 4])    ~>* Memory 3 [3, 2, 1] [4]
  focusRel (-2) (Memory 2 [2, 1] [3, 4]) ~>* Memory 0 [] [1, 2, 3, 4]
  ```

  Můžete předpokládat, že vstupní index bude v testech této funkce vždy validní pro vstupní paměť (tj. adresa bude v rámci paměti vždycky validní).

- `focusAbs :: Integer -> Memory a -> Memory a`

  Funkce, která pracuje stejně jako předchozí, ale na vstupu dostane absolutní pozici, na kterou se v rámci paměti máme zaměřit. Výsledkem je tedy paměť zaměřená na pozici odpovídající hodnotě prvního argumentu.

Všimněte se, že posun na absolutní pozici mimo jiné znamená, že výsledkem volání této funkce na dvou obsahově stejných pamětech s různým zaměřením budou dvě stejné paměti (jak obsahem, tak zaměřením).

```
focusAbs 2 (Memory 0 [] [1, 2, 3, 4]) ~>* Memory 2 [2, 1] [3, 4]
focusAbs 2 (Memory 3 [3, 2, 1] [4])   ~>* Memory 2 [2, 1] [3, 4]
```

Stejně jako v případě předchozí funkce můžete předpokládat, že vstupní adresa je v testech vždy validní.

- `getRegister :: Registers -> RegisterName -> Value`

  Funkce na vstupu dostane sadu registrů a název registru, jehož hodnotu chceme získat a tuto hodnotu vrátí.

  ```
  getRegister (Registers 42 66 42) R2 ~>* 66
  ```

- `setRegister :: Registers -> RegisterName -> Value -> Registers`

  Funkce na vstupu dostane sadu registrů, název registru a hodnotu, kterou chceme do registru uložit. Návratovou hodnotou je aktualizovaná sada registrů.

  ```
  setRegister (Registers 42 66 42) R2 42 ~>* Registers 42 42 42
  ```

- `evalStep :: Program -> Data -> Registers -> (Maybe Output, Program, Data, Registers)`

  Vstupem je program, data a sada registrů. Funkce `evalStep` vykoná jeden krok procesoru – jednu, právě zaměřenou instrukci – a výstup vrátí v podobě čtveřice. Pokud paměť už žádnou instrukci neobsahuje (jsme na konci paměti, není zaměřené žádné políčko), chovejte se, jako byste narazili na instrukci `Halt`. První složkou je možný výstup vyprodukovaný instrukcí, druhou složkou je nový stav programu. Jinak řečeno paměť instrukcí, která je zaměřená na následující (ne nutně následující ve smyslu pořadí v paměti, protože máme instrukce skoku) instrukci. V případě instrukce `Halt` zaměření neměňte, naopak v případě neplatné adresy při instrukci skoku opět zavolejte funkci `error` s rozumnou chybovou hláškou. Třetí složkou je nový stav dat a poslední je aktuální stav registrů.

  Nápověda: Doporučujeme implementovat podporu instrukcí podle skupin, do kterých je dělíme v příslušné části. Pro instrukce v rámci jedné skupiny si nejspíš budete chtít zavést i nějaké pomocné funkce, které vám umožní vyhnout se duplikaci kódu a usnadní vám práci.

- `eval :: [Instruction] -> [Value] -> [Output]`

  Funkce pro evaluaci programu. Nejprve ze seznamů instrukcí a hodnot vytvoří program (omezenou paměť instrukcí) a data (ničím neomezenou paměť hodnot, přičemž políčka mimo vstupní seznam budou mít výchozí hodnotu `0`). Dále vytvoří sadu registrů, přičemž všechny tři registry mají výchozí hodnotu `0`.

  Následně funkce postupně vykonává vstupní instrukce, dokud nenarazí na instrukci `Halt`, nebo nevykoná poslední instrukci programu. Výstupem funkce je seznam všech výstupů z **vykonaných** výstupních instrukcí v pořadí, ve kterém byly tyto výstupy programem produkovány.

  Tato funkce by měla svůj výstup počítat líně. Jinak řečeno, nemůžete předpokládat konečnost vstupních seznamů a zároveň očekáváme, že tato funkce bude produkovat výstup i na nekonečných vstupech a cyklících programech.

Popis instrukcí

Jak již bylo řečeno, všechny instrukce reprezentujeme datovým typem `Instruction`, který si můžete prohlédnout níže v celé jeho kráse. Instrukce dělíme do několika skupin podle toho, s čím pracují.

```
data Condition = Neg | Zero | Pos deriving (Show, Eq)

data Instruction = Move RegisterName RegisterName
                 | Assign RegisterName Value
                 | Add RegisterName RegisterName
                 | Negate RegisterName

                 | Load RegisterName
                 | Store RegisterName
                 | Focus AddressType RegisterName
```

```
    | Jump AddressType Integer
    | JumpIf Condition RegisterName AddressType Integer

    | Out RegisterName
    | Trace String

    | Halt
    deriving (Show, Eq)
```

a) instrukce pracující s registry

- `Move RegisterName RegisterName`

  Instrukce, která značí přesun/zkopírování hodnoty z druhého registru do prvního.

- `Assign RegisterName Value`

  Tato instrukce uloží do daného registru hodnotu svého druhého argumentu.

- `Add RegisterName RegisterName`

  Při této instrukci procesor sečte hodnoty obou registrů a výsledek vloží do prvního registru.

- `Negate RegisterName`

  Tato instrukce způsobí negaci hodnoty v odpovídajícím registru.

b) instrukce pracující s pamětí

- `Load RegisterName`

  Tato instrukce uloží do registru hodnotu uloženou v paměti na aktuálním indexu.

- `Store RegisterName`

  Tato instrukce uloží do paměti – na aktuální index – hodnotu uloženou v daném registru.

- `Focus AddressType RegisterName`

  Tato instrukce změní zaměření paměti procesoru na pozici určenou typem adresy a adresou, která je uložena v příslušném registru.

c) instrukce skoku

- `Jump AddressType Integer`

  Tato instrukce odpovídá nepodmíněnému skoku – procesor skočí na adresu v programu určenou oběma argumenty hodnotového konstruktoru.

- `JumpIf Condition RegisterName AddressType Integer`

  Při vykonávání této instrukce dostaneme v prvním argumentu podmínku, kterou musí hodnota v registru v druhém argumentu splnit, aby byl skok proveden. Tyto podmínky jsou tři, určené datovým typem `Condition`, přičemž `Neg` značí zápornou hodnotu v obsahu registru, `Zero` hodnotu nula a `Pos` kladnou hodnotu registru.

  V případě, že hodnota splňuje danou podmínku, provede se skok odpovídající chování procesoru na instrukci `Jump` s posledními dvěma argumenty stejnými jako u této instrukce. V opačném případě se žádný skok neprovede a program pokračuje následující instrukcí.

d) výstupní instrukce

- `Out RegisterName`

  Procesor při vykonávání této instrukce vyprodukuje hodnotu daného registru zabalenou v typu `Output`. Hodnotu vyprodukuje do první složky výstupní čtveřice v `evalStep`, respektive do výstupního seznamu v `eval`.

- `Trace String`

  Tato instrukce je obdobná předchozí instrukci, ale místo hodnoty registru procesor vyprodukuje zadaný řetězec.

e) speciální instrukce

- `Halt`

Instrukce `Halt` ukončí výpočet, a to i v případě, kdy program obsahuje nezpracované instrukce.

Poznámky a tipy

- Importovat smíte moduly z balíku `base`.[2]
- Neduplikujte kód! Snažte se vždy využít funkce, které jste již naprogramovali. Pokud to nejde přímo, ale přesto vidíte v řešení podobu, vytkněte podobnou část do pomocné funkce.
- Pomocné funkce definujte lokálně, pokud jsou využívány jedinou funkcí.
- Funkce jsou v kostře zadefinovány jako `undefined`, takže projdou překladem, ale jejich zavolání způsobí chybu.
- Nejste-li si jisti nějakou částí zadání, zeptejte se v diskusním fóru.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.
- Přebíráte-li kód odjinud, uveďte zdroj, jinak bude na vaši práci pohlíženo jako na plagiát.
- Než řešení odevzdáte, **pečlivě si přečtěte následující sekci** a ujistěte se, že váš kód splňuje všechny náležitosti. Neztrácejte body jen kvůli nepozornému čtení pokynů.

Odevzdání

Tento domácí úkol se neodevzdává přes odpovědník, nýbrž přes příslušnou odevzdávárnu v Informačním systému. O tom, kterou odevzdávárnu máte použít, rozhoduje číslo vaší seminární skupiny.

- Do odevzdávárny vkládejte **jediný** soubor s příponou `.hs` obsahující vaši implementaci **všech** požadovaných funkcí.
  - Pokud chcete odevzdat znovu, můžete soubor přepsat či přidat nový, nezáleží na tom – vyhodnocuje se vždy nejnovější odevzdaný soubor.
  - Žádné jiné soubory nevkládejte.
  - Již vložené soubory nepřejmenovávejte, mohou se pak vyhodnotit dvakrát.
- Vaše řešení **musí zachovat všechny definice datových typů tak, jak jsou v zadání**, jediná povolená modifikace je přidání instance typové třídy.
- Řešení musí jít přeložit překladačem GHC 8.10. Proto vám doporučujeme, abyste si před odevzdáním svůj kód zkusili zkompilovat a spustit na Aise (nezapomeňte přidat modul s novým GHC).
- **Všechny globální funkce musí mít typovou signaturu**.
- V odevzdaném souboru neuvádějte hlavičku `module` (pokud nevíte, o co se jedná, vůbec to nevadí).

Vyhodnocování

- Vyhodnocení po nahrání souboru **není** okamžité.
  - Automatický testovací nástroj kontroluje soubory v odevzdávárně v pravidelných intervalech několika minut. Podle času odevzdání a vytížení vyhodnocovacího serveru může vyhodnocení trvat několik desítek minut.
  - Pokud se vám výsledky neobjevili cca do půl hodiny a neblíží se zatím čas deadline, dejte nám vědět ve fóru.
- Po vyhodnocení se získané body a případný výpis neprošedších testů **objeví v poznámkovém bloku**.
  - U nesprávně implementovaných funkcí se dozvíte příklad vstupu, na němž se váš výsledek neshoduje s očekávaným.
  - Nejprve jsou v bloku uvedeny vstupy, následně výstup učitelského řešení a váš výstup.
  - V bloku uvidíte dvě sady testů – *build* a *test*, může se stát, že se zobrazí nejprve jen *build*, do pěti minut by pak měly přibýt výsledky samotných testů funkčnosti.
- Máte **pět možností odevzdání**, započítává se nejlepší z nich.

[2] Výjimku tvoří moduly, které jsou „Unsafe“, ty však určitě nebudete potřebovat.

  - Pokud při odevzdání neprojde kontrola syntaxe, tak se do tohoto limitu nepočítá, pokud však kontrola projde, je automaticky započítáno a nelze to zvrátit.
- Další odevzdání provedete tak, že do odevzdávárny nahrajete novou verzi.
- Vzhledem k prodlevám při vyhodnocování neodkládejte práci na poslední chvíli, ať možnost vícenásobného odevzdání v případě potřeby vůbec stihnete využít.
  - S blížícím se termínem uzavření odevzdáváren očekávejte větší (i několikahodinové) prodlevy.
  - Není žádná garance rychlosti vyhodnocování (může se tedy stát, že výsledky řešení odevzdaného v neděli ve 21.00 neuvidíte do konce deadline).

S odevzdávárnou zacházejte s rozvahou, abyste nepřišli o možnosti odevzdání. I když nahrajete nové řešení ještě před zveřejněním výsledku v poznámkovém bloku, vyhodnocovací nástroj už může mít (a pravděpodobně má) vaše dřívější odevzdání ve frontě. Z jeho pohledu tak došlo ke dvěma odevzdáním a vy si vyplýtváte jeden pokus. Podobně se vám mohou započítat odevzdání navíc, pokud do odevzdávárny omylem vložíte více než jeden soubor nebo pokud soubor přejmenujete (každý soubor se vezme jako samostatné odevzdání).

Testy

Stejně jako u předchozí velké domácí úlohy i tentokrát máte v kostře pár testů ke každé funkci. Formát testů zůstal zachován, neměli byste mít problém dopsat si své vlastní – stačí přidat další n-tice do příslušného seznamu.

Samotné testy potom můžete spustit zavoláním funkce `main` z interpretu (máte-li soubor již načtený) nebo pomocí příkazu `runhaskell 12post5.hs` z terminálu (v případě, že jste si naši kostru uložili pod jiným názvem, bude se správný příkaz lišit ve jménu souboru).

Assembler

Mimo testy máte v kostře úkolu ještě jednoduchou, ale užitečnou nadstavbu nad naším emulátorem procesoru, respektive nad jeho instrukcemi – Assembler.

Hlavním přínosem je existence návěstí, na která můžete v programu skákat. To je užitečné třeba v situaci, kdy máte program, který nefunguje, jak má, a vy si do něj chcete přidat nějaký výpis. Tím, že do seznamu instrukcí ale přidáte další, posunete absolutní adresy všech instrukcí po právě vložené o jedna. To vám pravděpodobně rozbije spoustu instrukcí skoku, které nyní budou skákat o políčko vedle.

Vezměme si například program pro výpočet Fibonacciho posloupnosti (který produkuje postupně celou posloupnost pomocí instrukce `Out`).

```
fibInst :: [Instruction]
fibInst = [ Assign      R2  1
          , Out         R1
          , Move        R3  R2
          , Add         R2  R1
          , Move        R1  R3
          , Jump  Absolute  1
          ]

fibEval :: [Output]
fibEval = eval fibInst []
```

Instrukce skoku, která je na konci seznamu instrukcí, počítá s tím, že instrukce pro výpočet Fibonacciho posloupnosti začínají první instrukcí. Pokud bychom si třeba (velice uměle) chtěli na začátku výpočtu vypsat, že začínáme počítat Fibonacciho posloupnost (například pokud bychom byli uvnitř nějakého většího programu), mohli bychom to udělat instrukcí `Trace`.

```
fibInst = [ Trace "Begin to compute Fibonacci sequence."
          , Assign      R2  1
          , Out         R1
          , Move        R3  R2
          , Add         R2  R1
          , Move        R1  R3
```

```
          , Jump  Absolute  2
          ]
```

Všimněte si, že s přidáním `Trace` jsme museli změnit i adresu absolutního skoku.

Takové změny jsou ve větších programech problematické. S využitím assembleru vás to trápit nemusí a psaní v něm se příliš neliší od psaní samotných instrukcí. Ekvivalentní program pro výpočet Fibonacciho posloupnosti vypadá takto:

```
fibAsm :: [Assembly]
fibAsm = [ trace "Begin to compute Fibonacci sequence."
         , assign      R2  1

         , "fib" :>
           out         R1
         , move        R3  R2
         , add         R2  R1
         , move        R1  R3
         , jump  Absolute  "fib"
         ]
```

přičemž spuštění odpovídajícího výpočtu vyžaduje pouhé jedno volání funkce navíc.

```
fibEval :: [Output]
fibEval = eval (assemble fibAsm) []
```

Jak můžete vidět, instrukce v assembleru se liší pouze tím, že jejich názvy píšeme s malým prvním písmenem (nejde o hodnotové konstruktory, ale o funkce) a tyto assemblerové instrukce následně musíme přeložit voláním funkce `assemble`. Návěstí pojmenováváme řetězcem (`String`) a mezi toto jméno a první instrukci odpovídajícího bloku vložíme operátor `(:>)` – všimněte si, že před `out` není čárka. Poslední instrukce skoku potom neskáče na konkrétní adresu, ale na přidané návěstí `"fib"`, které značí začátek instrukcí počítající další číslo posloupnosti.

V kostře najdete ještě několik dalších zajímavých programů a pomocných funkcí, kde se můžete dále inspirovat a použít je k testování. Zároveň na konci souboru najdete kód řešící samotný assembler. Tento kód přesahuje výrazně rozsah tohoto kurzu a jeho čtení vám tedy nejspíš nic nedá.

Všechny testy, které používáme, pracují s instrukcemi, tento assembler je pouze pro vás, aby se vám případné testovací prográmky psaly lépe.

Hodnocení

Za funkčnost můžete od automatických testů obdržet **až 2,5 bodu** podle toho, které funkce (nebo jejich části) se vám podařilo správně implementovat (viz tabulka níže). Již tradičně počítejte s tím, že za neošetřené okrajové případy žádné body nedostanete – částečné body lze získat jen u funkcí `evalStep` a `eval` a to přesně v rozsahu daném následující tabulkou.

Funkce	Body
`fromList`	0,1
`fromListInf`	0,1
`getRegister`	0,05
`setRegister`	0,05
`focusRel`	0,2
`focusAbs`	0,2
`evalStep` s instrukcemi skupiny a)	0,2
`evalStep` s instrukcemi skupin a) a b)	0,2
`evalStep` s instrukcemi skupin a) a c)	0,3
`evalStep` se všemi instrukcemi	0,6
`eval`	0,3
`eval` – lenost	0,2

Následně vám cvičící dají zpětnou vazbu na kód, a také vám za úhlednost a pochopitelnost řešení mohou udělit dalšího 0,5 bodu. Snažte se proto kód psát hezký a případné neočividné, či zajímavé části řešení stručně komentujte v kódu.

Tipy k psaní hezkého kódu naleznete ve sbírce. Hodnotit na kvalitu se bude poslední odevzdání s maximem bodů ze všech vašich odevzdání. Pokud vám tedy po dosažení plného počtu automaticky přidělovaných bodů ještě zbývají pokusy, můžete bezpečně zkusit svůj kód zkrášlit.

V součtu tedy můžete za úlohu získat **až 3 body**.