# IB016 – 2. velký úkol – Cejtinův převod

### Termíny

- Úterý **18.5.** 2021 23.59 – implementační část
- Úterý **25.5.** 2021 23.59 – peer review

Tento úkol bude o výrokových formulích, Cejtinově převodu a monádách RWS.

Kostru řešení s testy naleznete ve studijních materiálech.

Vaším úkolem bude naprogramovat několik funkcí pracujících s výrokovými formulemi. Pro formule tentokrát nemáme jednu reprezentaci, ale hned několik. Abychom s nimi mohli pohodlně pracovat, definujeme typovou třídu `Formula`, jejíž instancemi budou všechny naše reprezentace.

```
class Formula f where
    eval :: (MonadReader Valuation m) => f -> m Bool
    getVariables :: f -> Set Varname
```

Typová třída `Formula` definuje dvě funkce: `eval` a `getVariables`. První funkce pracuje v monádě čtenáře, který si udržuje valuaci proměnných, a na vstupu dostane formuli, jejíž valuaci vrátí. Valuaci reprezentujeme naším datovým typem `Valuation`, který je jen `newtype` obalující `Map Varname Bool`, kde `Varname` je typovým aliasem pro `String`. V případě, že nějaká volná proměnná ve formuli neexistuje v zadané valuaci, její implicitní hodnotou je `False`.

Například

```
runReader (eval (Var "x")) (Valuation Map.empty) ~>* False
runReader (eval (Var "x")) (Valuation (Map.fromList [("x", False)] ~>* False]
runReader (eval (Var "x")) (Valuation (Map.fromList [("x", True)]  ~>* True]
```

Druhá funkce, `getVariables` pak pro danou formuli vrátí množinu volných proměnných v dané formuli.

```
getVariables (Var "x") ~>* fromList ["x"]
getVariables (Not (Var "x")) ~>* fromList ["x"]
```

Vaším úkolem je implementovat dvě instance této typové třídy a to pro dva datové typy, kterými formule reprezentujeme. První datový typ je `FormulaTree`, který reprezentuje klasickou výrokovou formuli, a `CNF`, který reprezentuje formuli v konjuktivní normální formě.

Kromě výše uvedených funkcí budete muset definovat ještě funkci `toCNF`, která na vstupu dostane formuli `FormulaTree` a vrátí k ní *ekvisplnitelnou* formuli v konjuktivní normální formě. To znamená, že výstupní formule je splnitelná právě tehdy, když je splnitelná i vstupní formule. Pro převod využijte Cejtinův převod, což je algoritmus, při kterém dochází k (pouze) lineárnímu nárůstu velikosti vstupní formule. Pěkně popsaný jej najdete na anglické wikipedii. Pro jména nových proměnných v rámci výstupní formule můžete použít prefix `"aux"`, aby nedošlo ke konfliktům.

*Nápověda: Důrazně doporučujeme pro implementaci této funkce využít monádu RWS; významně si tak usnadníte práci a váš kód bude jistě čitelnější.*

Poslední funkcí k implementaci je funkce `eval'`, která na vstupu dostane formuli a valuaci a vrátí hodnotu dané vstupní formule při dané valuaci. Všimněte si, že díky typové třídě bude tato funkce fungovat pro všechny instance typové třídy (a tedy i pro ty, které bychom si případně definovali v budoucnu).

```
eval' (Not (Var "x")) (Valuation M.Empty) ~>* True
eval' (Not (Var "x")) (Valuation (M.fromList [("x", True)])) ~>* False
eval' (Var "x") (Valuation (M.fromList [("x", True), ("y", True)])) ~>* True
```

## Datové typy

Jak bylo řečeno, pro reprezentaci formulí máme dva datové typy. První je `FormulaTree`, který vypadá následovně:

```haskell
data FormulaTree = Var Varname
                 | Not FormulaTree
                 | BinOp Op FormulaTree FormulaTree
                 | Let Varname FormulaTree FormulaTree
                 deriving ( Eq, Show, Read )
```

přičemž `Var` značí proměnnou jménem `Varname`, `Not` negaci odpovídající formule a `BinOp op` aplikaci binární spojky na dvě formule. Konstruktor `Let` definuje novou proměnnou jména `Varname` jako ekvivalentní formuli v druhém argumentu v rámci třetí formule. Takto definovaná proměnná je vázaná, takže se nesmí objevit ve výstupu z funkce `getVariables`.

```haskell
getVariables (Let "a" (Var "x") (BinOp And (Var "a") (Var "y"))) ~>* fromList ["x","y"]
getVariables (BinOp And (Var "x") (Let (Var "x") (Var "a") (Var "x"))) ~>* fromList ["x", "a"]
```

Druhým datovým typem je `CNF`, který je zdánlivě složitější:

```haskell
data Literal = Pos Varname | Neg Varname deriving ( Eq, Show, Read )

newtype Clause = Clause { getLiterals :: [Literal] }
               deriving ( Eq, Show, Read, Semigroup, Monoid )

newtype CNF = CNF { getClauses :: [Clause] }
            deriving ( Eq, Show, Read, Semigroup, Monoid )
```

Jak je z definice patrné, `CNF` je seznamem klauzulí, přičemž klauzule je seznamem literálů. Literál je pak proměnná, nebo její negace. V konjuktivní normální formě jsou klauzule spojeny konjunkcí, literály v klauzuli jsou potom v disjunkci. Tedy hodnotě

```haskell
CNF {getClauses = [Clause {getLiterals = [Pos "x",Neg "y"]},
                   Clause {getLiterals = [Pos "z"]}
                  ]}
```

odpovídá formule `(x || !y) && z`.

## Kostra

Ve studijních materiálech najdete kostru, ve které je opět několik testů (určitě si napište i nějaké vlastní) a tentokrát v kostře najdete i několik užitečných funkcí, které by vám při implementaci mohly pomoci. Především se jedná o typovou třídu `Pretty` a s ní svázanou funkci `pprint :: f -> String`, která vrací hezčí (a přehlednější) textovou reprezentaci vstupní formule.

## Odevzdávání

- Svou implementaci odevzdávejte do příslušné odevzdávárny v ISu.
- Na odevzdávání máte neomezeně mnoho pokusů, výsledky testů naleznete v poznámkových blocích, jak jste zvyklí z IB015.
- Po ukončení odevzdávání budou zveřejněna řešení všech studentů a zároveň vám budou přiděleni dva spolužáci, na jejichž implementaci následně napíšete peer review, tedy recenzi jejich kódu.
- Peer review se píší ve formě komentářů do recenzovaného zdrojového kódu a odevzdávají se na Aise. K tomuto ještě obdržíte e-mail s pokyny.

### Technické požadavky

- Odevzdávejte jediný soubor s příponou `.hs`.
- Řešení musí být přeložitelné GHC 8.10. Můžete si to vyzkoušet na Aise s modulem `ghc`.
- Všechny globálně definované funkce musí mít typovou signaturu.
- Řešení nesmí obsahovat hlavičku modulu.

### Moduly a rozšíření

- Můžete využít libovolné moduly z balíků `base` a `mtl`.
- Povolenými rozšířeními jsou také `PatternGuards`, `LambdaCase` a `TupleSections`.

## Hodnocení

Za úlohu můžete získat až čtyři body. Pro úspěšné absolvování předmětu potřebujete alespoň jeden bod (vizte IS pro podrobné informace). Doporučujeme tedy řešení úlohy zbytečně neodkládat.

V tomto semináři nerozlišujeme body za krásu a body za funkčnost, proto berte projití testy pouze jako potvrzení toho, že něco neděláte úplně špatně, ne jako signál, že máte 4 body v kapse. Řešení, která budou funkční, ale ošklivá, si plný počet nezaslouží. V žádném případě byste se neměli dopouštět prohřešků, před nimiž varujeme už v IB015.

Body vám udělí cvičící i se zpětnou vazbou až poté, co budou odevzdána i peer review.

Řešení tentokrát budou veřejná, takže se na ně po termínu odevzdání bude moci kdokoliv podívat. To proto, aby bylo jednodušší psát si vzájemně peer review, a částečně vás tím chceme motivovat k psaní hezčího kódu. Nebojte se pomoci si nástrojem HLint (vaši recenzenti to určitě udělají).

## Poznámky a tipy

- Nechte se vést typy. Využívejte typované díry!
- Neduplikujte kód! Využívejte vlastních i knihovních funkcí. Pomoci vám v tom může Hoogle.
- Pokud vám není něco jasné, zeptejte se v diskuzním fóru nebo na ircovém/discordovém kanále.
- V případě, že přebíráte kód odjinud, uveďte zdroj.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.