Programování v Haskellu

KSI MFF CUNI CZ

2021

verze slajdů: `v2020-39-gbff1650`
sestavení: `2021-12-12 20:42:07 +0100`

Úvod a předběžnosti

Meta, požadavky

NPRG068, 2/0 Zk, 3 kredity

kratochvil@ksi.mff.cuni.cz

http://ksi.mff.cuni.cz/~kratochvil/haskell

Výuka asi bude nejspíš probíhat prezenčně.

V případě potíží přejdeme na distanční.

Požadavky

1. Domácí úkoly — využití prostředků jazyka
   - rozsah cca. 5% LOC úkolů z Javy a C++
   - Zadání budou na webu
   - Předběžný rozvrh:

Úkol	Zadání	Termín	(Týdny)
1	30. 9.	28. 10.	1–5
2	28. 10.	25. 11.	5–9
3	25. 11.	6. 1.	9–14

2. Zkouška — max 5–10 minut lehce teoretické diskuze
   - zhodnocení odevzdaných úkolů
   - témata budou vyvěšená na webu
   - témata $\subset$ obsah slajdů

Co je potřeba do kdy stihnout?

- **Odevzdat úkoly v termínu.**
  Termín je do půlnoci dne označeného jako termín (tj. první úkol byste měli začít odevzdávat nejpozději 28. 10. ve 23:59:59.999).
  Pokud nebudete stíhat, **dejte vědět předem**.
- **Jít na zkoušku.** (alternativně: dojít k počítači se zůmem)
  Termíny budou v SISu.
  Výsledné hodnocení bude odvozené především od množství a kvality odevzdaných úkolů, zkoušku berte jako kontrolu autentičnosti vašich řešení (a příležitost se dozvědět něco zajímavého, co vám v průběhu semestru uniklo). Obtížnost by *po zvládnutí DÚ* měla být naprosto zanedbatelná.

DÚ — požadavky

- **Velikost řešení** není vhodné přehánět (čím menší, tím lepší!)
- **Správnost**:
  - Rozumné použití typového systému pro prevenci běhových chyb
  - Rozumná časová složitost použitých algoritmů

DÚ — odevzdávání

1. Z řešení vyrobte `cabal`ový balík
2. Použijte `cabal sdist`
3. Výsledek nahrajte do odpovídající kolonky v SISu.

Slajdy

Slajdy jsou volně k dispozici na webu (budou přibývat přes semestr)

Pokud najdete chybu nebo vymyslíte zlepšení:

- řekněte si o git access (implementace: `lhs2TeX` + beamer)
- malé věci reportujte mailem

Typické zlepšení: ilustrativní obrázek obsahující kočku.

Obsah přednášky

1. opakování z neprocedurálního programování
2. šťourání v typech, monádách a IO
3. standardní knihovna
4. implementace jazyka, RTS, typového systému a modularizace
5. nejužitečnější ostatní abstrakce a knihovny

Proč sem chodit?

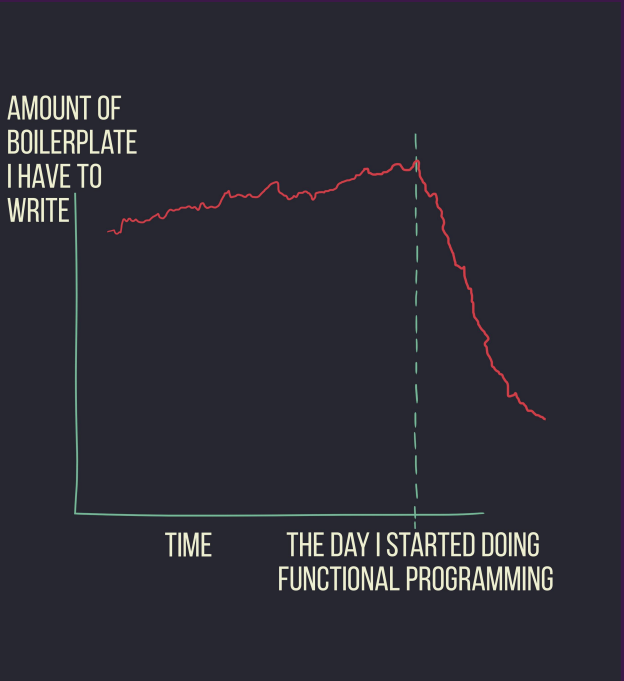
- návrhové vzory z funkcionálního programování a Haskellu jsou jedny z nejmladších a nejužitečnějších věcí, které komunita má
- haskellové programy nepadají
  (tj.: haskellové programy nepadají z hloupých důvodů)
- funkcionální programování je Opravdu Hodně Expresivní
- důsledek: průmysl začíná chtít funkcionální programátory
  (ještě se musí zjistit, že Scala je fakt úlet)
- zjistíte i něco o ostatních jazycích

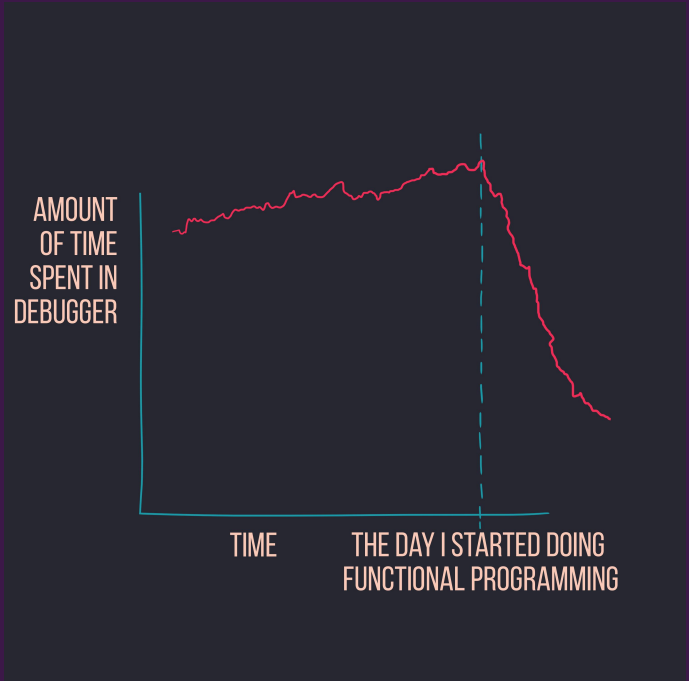
Proč sem chodit?

- Bonus 1: Haskellová praxe dramaticky zlepšuje schopnost programovat bez chyb ve všech ostatních jazycích.

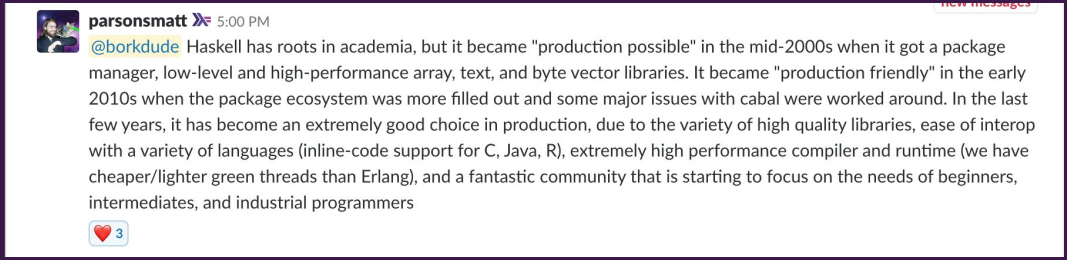
  Programátor umí z libovolného typecheckeru vyždímat maximum.

- Bonus 2: Hodně temných a těžko vysvětlitelných jevů z C++ má v Haskellu svou pochopitelnější variantu.


AMOUNT OF
BOILERPLATE
I HAVE TO
WRITE
TIME
THE DAY I STARTED DOING
FUNCTIONAL PROGRAMMING


AMOUNT OF TIME SPENT IN DEBUGGER
TIME
THE DAY I STARTED DOING FUNCTIONAL PROGRAMMING

Stručná historie

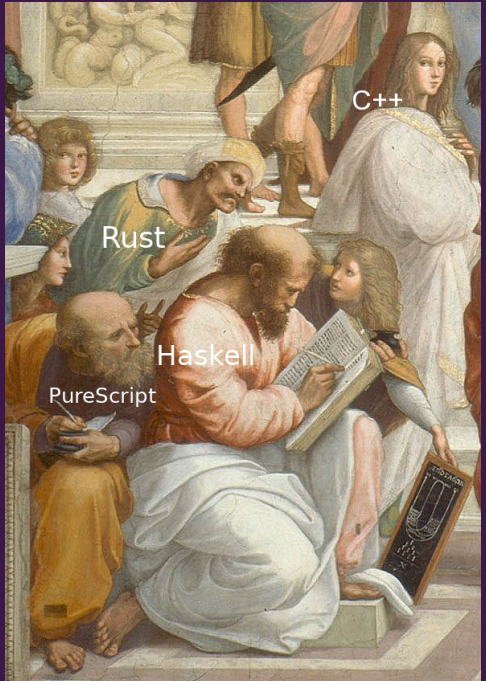


parsonsmatt 5:00 PM

@borkdude Haskell has roots in academia, but it became "production possible" in the mid-2000s when it got a package manager, low-level and high-performance array, text, and byte vector libraries. It became "production friendly" in the early 2010s when the package ecosystem was more filled out and some major issues with cabal were worked around. In the last few years, it has become an extremely good choice in production, due to the variety of high quality libraries, ease of interop with a variety of languages (inline-code support for C, Java, R), extremely high performance compiler and runtime (we have cheaper/lighter green threads than Erlang), and a fantastic community that is starting to focus on the needs of beginners, intermediates, and industrial programmers

❤️ 3

Industry-ready

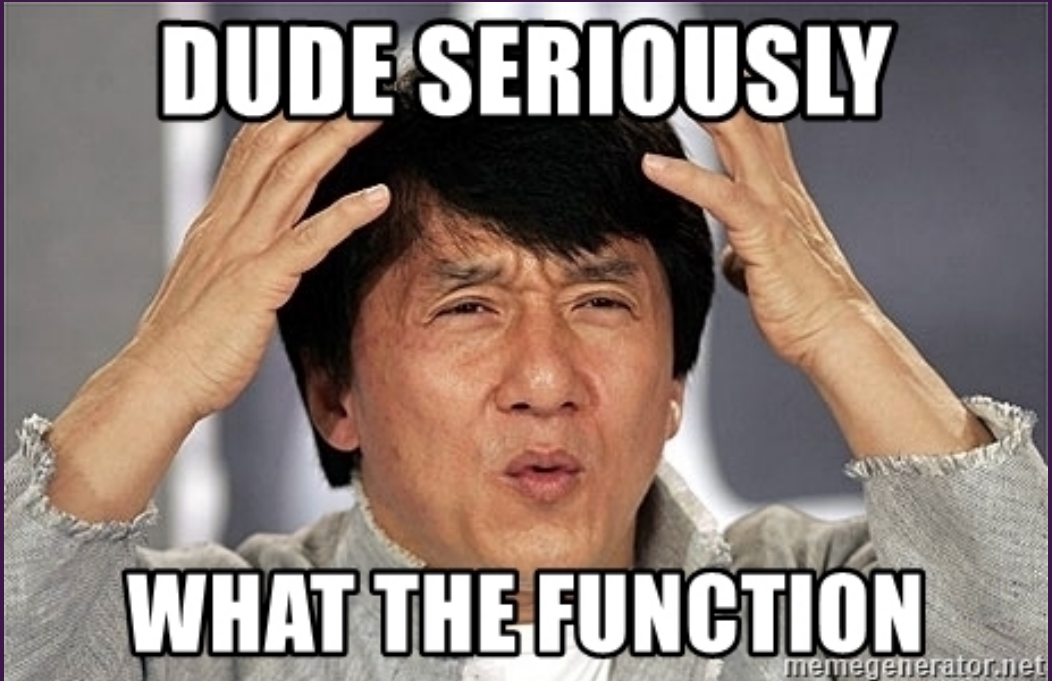

C++
Rust
Haskell
PureScript

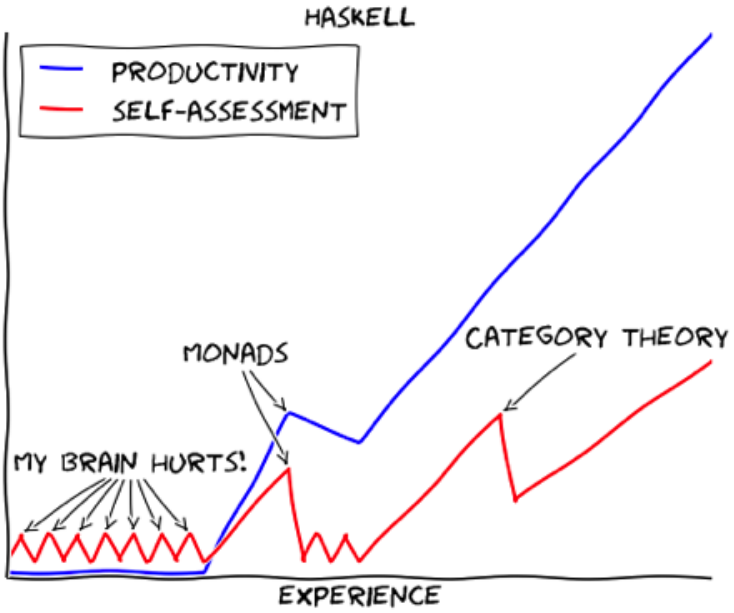
Expresivní!

$$wordCount = length \cdot words \langle\$\rangle readLine$$

$$rle = map\ (liftA2\ (,)\ head\ length) \cdot group$$

$$scc\ g = dfs\ g \cdot reverse \cdot postOrd\ \$\ transposeG\ g$$


DUDE SERIOUSLY
WHAT THE FUNCTION
memegenerator.net


HASKELL
PRODUCTIVITY
SELF-ASSESSMENT
MONADS
CATEGORY THEORY
MY BRAIN HURTS!
EXPERIENCE

Literatura

- Learn You A Haskell For Great Good
  `http://learnyouahaskell.com/`
- Real World Haskell
  `http://book.realworldhaskell.org/`
- Hoogle
  `https://hoogle.haskell.org/`
- Typeclassopædia
- Haskell Wikibook
- What I Wish I Knew When Learning haskell
  `https://github.com/sdiehl/wiwinwlh`

⋮

Okolní předměty

- NPRG005 Neproc
  - typový systém Haskellu je skoro Prolog!
  - hlavním účelem NPRG005 je naučit se rekurzi, tady budeme trochu blíž normálnímu programování
- NAIL078, NAIL079 Lambda kalkulus a funkcionální programování
  - (ne)rozhodnutelnost, typové systémy, souvislost s logikou
- NAIL097 Funkcionální programování
  - sémantika, typové systémy víc zblízka
  - občas Agda a kategorie

Okolní jazyky

- Haskell je ‘managed’ — o napojení na systém se stará run-time.
  - Nelze s ním (úplně) nahradit nízkoúrovňové jazyky (C, C++, Rust).
  - Je velmi vhodné naplno využít potenciál run-time vrstvy a Haskellem nahrazovat různé polovičaté C-like managed jazyky — typicky Javu a C#, běžně se stonásobnou úsporou kódu, času, chyb, rychlosti a nervů.
- Haskell je ‘typovaný’ a ‘kompilovaný’ — nelze s ním (úplně) nahradit shell-like jazyky (Bash, R, K, …), ale jde se k tomu dost přiblížit.
- Haskell není ‘browserový’, ale jde (s trochou opatrnosti) kompilovat do JavaScriptu.

Aktuální obsah slajdů

1. Úvod a předběžnosti
2. Rychloúvod do funkcionálního programování
3. Praxe s monádami: Parsovací kombinátory
4. Balíčky a knihovny
5. Tour de Prelude
6. Kontejnery
7. Kompilujeme Haskell
8. Curry-Howardova korespondence
9. Optika, čočky a hranoly
10. Transformujeme monády
11. Stringologie a formátování textu
12. IO!
13. Grafické knihovny
14. Debugování, testování, benchmarking
15. Webové aplikace
16. eDSL a interprety

Rychloúvod do funkcionálního programování

Kde sehnat prostředí?

Použijte ghcup

`https://github.com/haskell/ghcup-hs`

Kde sehnat prostředí (bez ghcup)

Windows/Mac: Nainstalujte si Haskell Platform.

- `https://www.haskell.org/platform/`

Unix-compatible: Použijte oblíbený balíčkovač.

- `apt-get install ghc`
- `yum install ghc`
- `pkg install ghc`
- ...
- ArchLinux: `ghc-static`, `cabal-static`!

Ve vlastním zájmu používejte nějaký unix.

Pozor: Ubuntu se nepočítá jako unix.

Hello

```
$ cat > hello.hs <<EOHS
main = putStrLn "Hello MFF!"
EOHS
$ runhaskell hello.hs
Hello MFF!
$ ghc hello.hs -o hello
[1 of 1] Compiling Main  ( hello.hs, hello.o )
Linking hello ...
$ ./hello
Hello MFF!
```

Interpreter

```
$ ghci hello.hs
GHCi, version 8.2.2
[1 of 1] Compiling Main  ( hello.hs, interpreted )
Ok, one module loaded.
*Main> :t main
main :: IO ()
*Main> main
Hello MFF!
*Main> 5+5
10
```

Integrace s nástroji

Standardní pomůcky: `hindent`, `hlint`, `ghcid`.

Editory:

- obecně: haskell-language-server, ghc-mod
- vim: Haskell-vim-now, Syntastic, neco-ghc
- emacs: spacemacs + haskell-mode

IDE škodí kvalitě kódu. Pokud váš kód nebude číst nikdo jiný a bez IDE neumíte žít, můžete zkusit:

- leksah
- pluginy do IntelliJ, Eclipse, KDevelop

Syntax

Haskell je deklarativní jazyk, program je hromada definovaných hodnot:

$$\textit{jmenoHodnoty} = \textit{definiceHodnoty}$$

$$\textit{hodnotaSParametrem parametr} = \textit{definice}$$

Pokud je v programu hodnota `main`, runtime ji pro "spuštění" programu "vyhodnotí".

$$\textit{main} = \textit{print}\ (1 + 1)$$

Syntax výrazů

- Jména proměnných začínají malým písmenem: `a` `b` `ahoj`
- Literály: 123, 1.5, `'a'`, `"nazdar"`
- 2 výrazy za sebou = aplikace funkce na parametr:
  - *negate* 5
  - *take* 3 `"kocka"`
- operátory mají nižší prioritu než aplikace!
  - +, - ,++, ==, >=, <=, …
  - *negate* $5 - 10 \quad \rightsquigarrow$?
- kulaté závorky:
  - *take* $1 + 2$ `"kocka"`
  - *take* $(1 + 2)$ `"kocka"`

Operátory

Vlastní operátory jde definovat s asociativitou a prioritou:

infixr 3 $+\!\!+$
infixl 6 $/\%/$
infix 5 $:=>$

Operátor jde předělat na funkci uzávorkováním:

$5 + 10 \quad \rightsquigarrow 15$
$(+)\ 5\ 10 \quad \rightsquigarrow 15$

Identifikátor ve zpětných apostrofech je operátor:

mod $10\ 3 \quad \rightsquigarrow 1$
10 ‘*mod*‘ $3 \quad \rightsquigarrow 1$

Pozor na unární mínus.

Syntax funkcí

Funkci jde definovat jako definici s parametrem:

$$f\ a\ b\ c = b\hat{}2 - 4 * a * c$$

nebo ekvivalentně jako lambdu:

$$f = \lambda a\ b\ c \to b\hat{}2 - 4 * a * c$$

nebo ekvivalentně úplně rozdrobit:

$$f = \lambda a \to \lambda b \to \lambda c \to b\hat{}2 - 4 * a * c$$

- pojmenovávat lambdy není nutné
- λ se píše jako zpětné lomítko

Lokální definice

Občas se hodí mít něco zadefinovaného lokálně.

```
solve a b c =
  let d = b^2 - 4 * a * c
      solution op = (-b `op` sqrt d) / (2 * a)
  in (solution (+), solution (-))
```

To samé obráceně:

```
solve a b c = (solution (+), solution (-))
  where d = b^2 - 4 * a * c
        solution op = (-b `op` sqrt d) / (2 * a)
```

Řízení výpočtu

if *cond* **then** *a* **else** *b*

- *cond* musí být booleovská hodnota
- "if bez else" nedává matematicky smysl (jaký typ má nic?)

case *x* **of**
moznost1 $\rightarrow$ *vysledek1*
moznost2 $\rightarrow$ *vysledek2*
$\_ \rightarrow$ *necoJineho*

Řízení výpočtu 2

Pattern matching:

$$
\begin{aligned}
&\mathit{fac}\ 0 = 1 \\
&\mathit{fac}\ n = n * \mathit{fac}\ (n-1)
\end{aligned}
$$

Guards:

$$
\begin{aligned}
&\mathit{gcd}\ x\ y \\
&\quad \mid x < y \quad\quad = \mathit{gcd}\ y\ x \\
&\quad \mid y \leqslant 0 \quad\quad = x \\
&\quad \mid x > y \quad\quad = \mathit{gcd}\ y\ (x\ \text{`}\mathit{mod}\text{`}\ y) \\
&\quad \mid \mathit{otherwise} = \bot \quad \text{-- error}
\end{aligned}
$$

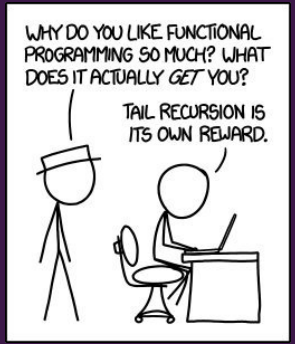
Kritické vlastnosti jazyka

- **Haskell je líný!** Prakticky žádnou funkci nevyhodnotí, dokud to nebude přímo potřeba.
  Důsledek: nekonečná data jsou OK
  allIntsFrom $n = n : \mathit{allIntsFrom}\ (n + 1)$
- **Haskell je referenčně transparentní!** Funkce se chovají ‘čistě’ a matematicky.
  Důsledek: kompilátor může kód upravovat výrazně volněji
  $f\ x + f\ x \equiv \textbf{let}\ \mathit{temp} = f\ x\ \textbf{in}\ \mathit{temp} + \mathit{temp} \equiv 2 * f\ x$
- **Haskell je staticky typovaný!**
  Důsledky:
  - při běhu nevznikne chyba z typových důvodů
  - znalost typů lze při překladu různě využívat

Cykly jsou ve funkcionálním jazyce zbytečné.

Cykly jsou ve funkcionálním jazyce zbytečné.

(jdou plně nahradit rekurzí)



Typy

Názvy jednoduchých typů začínají velkým písmenem, typy je možné specifikovat pomocí čtyřtečky.

$$\begin{aligned} &\mathit{cislo} :: \mathit{Integer} \\ &\mathit{cislo} = 5 \\ &\mathit{jineCislo} = (5 :: \mathit{Float}) \end{aligned}$$

V `ghci` je možné typ čehokoliv zjistit pomocí `:t`.

Typy funkcí

$$id :: a \to a$$

...ve skutečnosti: $id :: (\forall a)\ a \to a$

tj. neplatí $id :: Int \to a$ ani $id :: a \to Int$.

$$(,) :: a \to b \to (a, b)$$
$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$head :: [a] \to a$$
$$(+) :: Num\ a \Rightarrow a \to a \to a$$

...tzn. $(+) :: (\forall a \in \mathrm{Num})\ a \to a \to a$

Monomorfní:

$$id :: Int \to Int$$
$$gcd :: Integer \to Integer \to Integer$$

Curry-style parametry

Víceparametrové funkce neexistují!

$$f :: Int \to Int \to Int \to Int$$
$$f\ 1\ 2\ 3 :: Int$$

$$f :: Int \to (Int \to (Int \to Int))$$
$$f\ 1 :: Int \to (Int \to Int)$$
$$(f\ 1)\ 2 :: Int \to Int$$
$$((f\ 1)\ 2)\ 3 :: Int$$

Související kombinátory:

$$curry :: ((a, b) \to c) \to (a \to b \to c)$$
$$uncurry :: (a \to b \to c) \to ((a, b) \to c)$$

Haskell Curry

Curry-style parametry

Koncept porodil Schönfinkel (1924), v kombinátorové logice populárně použil až Curry.

Samozřejmě k dispozici je i varianta `schön`, `unschön`.

Moses Schönfinkel

“Arity pls.”

Kontrolní otázka: Kolik parametrů má funkce $id :: a \to a$?

Kontrolní otázka: Kolik parametrů má funkce *id* $::$ $a \to a$?

Odpověď: počty parametrů jsou blbost!

id 5
id gcd $10\ 15$
id id gcd $10\ 15$

Některé jednoduché typy

- `1` :: *Int*, *Integer*, *Word*, *Int64*, …
- `'c'` :: *Char*, *Word8*, …
- `1.23` :: *Float*, *Double*
- *True* :: *Bool*, *False* :: *Bool*
- `()` — existující nicneříkající typ
- *Void* — neexistující nicneříkající typ

Některé parametrizované typy

- $[a]$ — seznam
  $[1, 2, 3] :: \mathit{Num}\ a \Rightarrow [a]$
- **type** *String* $=$ $[\mathit{Char}]$
- *Maybe a*
  - $[\mathit{Just}\ \texttt{'c'}, \mathit{Nothing}] :: [\mathit{Maybe}\ \mathit{Char}]$

Definice algebraických datových typů

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$
$$\textbf{data}\ () = ()$$
$$\textbf{data}\ \mathit{Void}$$

Typy s obsahem:

$$\textbf{data}\ \mathit{DvojiceIntu} = \mathit{Par}\ \mathit{Int}\ \mathit{Int}$$

Konstrukce:

$$\mathit{Par}\ 2\ 3 :: \mathit{DvojiceIntu}$$

Destrukce (a.k.a. pattern match):

$$\mathit{f} :: \mathit{DvojiceIntu} \to \mathit{Int}$$
$$\mathit{f}\ (\mathit{Par}\ \mathit{a}\ \mathit{b}) = \mathit{b}$$

Parametrické typy

Parametry jako u funkcí (jen na jiném univerzu):

```
data Maybe a = Nothing | Just a
data (,) a b = (,) a b
data Either a b = Left a | Right b
data (->) a b    -- internal
```

Maybe, $(,)$ a $(\rightarrow)$ jsou *typové konstruktory* (v jazyce typů).

Nothing, *Just* a $(,)$ jsou *datové konstruktory* (v jazyce termů).

Konvence: Typové a datové konstruktory začínají velkým písmenem. Typové a datové proměnné začínají malým písmenem. (účel: přehlednost kódu, zjednoznačnění některých konstrukcí)

Parametrické typy

data *Dvojice a* = *Dva a a*
Dva 1 2 :: *Dvojice Int*
Dva (*Just* 5) *Nothing* :: *Dvojice* (*Maybe Int*)
Dva True 3 -- type error

Seznam

$$\textbf{data}\ [a] = (a : [a]) \mid [\,]$$

...přeloženo:

$$\textbf{data}\ [\,]\ a = (:)\ a\ ([\,]\ a) \mid [\,]$$

$$1 : 2 : 3 : [\,] \equiv [1, 2, 3]$$

Kind

Jak se zkontroluje správné použití typů a jejich parametrů?

Typům typů se říká Kinds (‘druhy’). Všechny typy hodnot patří do druhu $*$.

$$
\begin{aligned}
&\mathit{Bool} :: * \\
&\mathit{Int} :: * \\
&\mathit{Maybe} :: * \to * \\
&\mathit{Maybe\ Int} :: * \\
&[\,] :: * \to * \\
&(\to) :: * \to * \to * \\
&\mathit{RWST} :: * \to * \to * \to * \to (* \to *) \to * \\
&\mathit{Num} :: * \to \mathit{Constraint}
\end{aligned}
$$

V `ghci` se zjistí pomocí `:k`.

Ostatní definice typů

Typová synonyma (obě strany jsou ekvivalentní):

$$\textbf{type}\ \mathit{DvaInty} = (\mathit{Int}, \mathit{Int})$$

“Unboxed data” (typ se zkontroluje a pak se odmaže):

$$\textbf{newtype}\ \mathit{DvaInty} = \mathit{DvaInty}\ (\mathit{Int}, \mathit{Int})$$

Haskellové typy v C++ (velmi přibližně)

data $X\ a = X\ a\ a \mid Y\ a\ Int$

```
template<typename a> struct X {
  enum {X,Y} _variant;
  union {
    struct {a fld1; a fld2;} X;
    struct {a fld1; int fld2;} Y;
  };
};
```

Haskellové typy v C++ (velmi přibližně)

data *X a* = *X a a* | *Y a Int*

```
template<typename a> struct X {
  enum {X,Y} _variant;
  union {
    struct {a fld1; a fld2;} X;
    struct {a fld1; int fld2;} Y;
  };
};
```

newtype nemá značku, tj. nemůže rozlišovat mezi variantami.

Výběr z Prelude

$$fst :: (a, b) \to a$$
$$snd :: (a, b) \to b$$
$$show :: Show\ a \Rightarrow a \to String$$
$$read :: Read\ a \Rightarrow String \to a$$
$$putStrLn :: String \to IO\ ()$$
$$getLine :: IO\ String$$
$$print :: Show\ a \Rightarrow a \to IO\ ()$$
$$readLn :: Read\ a \Rightarrow IO\ a$$
$$(\$) :: (a \to b) \to a \to b$$
$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$

Výběr ze Prelude — seznamy

$head, last :: [a] \to a$
$tail, init :: [a] \to [a]$
$(+\!\!+) :: [a] \to [a] \to [a]$
$(!!) :: [a] \to Int \to a$
$elem :: Eq\ a \Rightarrow [a] \to a \to Bool$
$length :: [a] \to Int$
$take, drop :: Int \to [a] \to [a]$
$takeWhile, dropWhile :: (a \to Bool) \to [a] \to [a]$
$lines, words :: String \to [String]$
$unlines, unwords :: [String] \to String$
$map :: (a \to b) \to [a] \to [b]$
$filter :: (a \to Bool) \to [a] \to [a]$
$foldl :: (a \to x \to a) \to a \to [x] \to a$
$foldr :: (x \to a \to a) \to a \to [x] \to a$
$lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$

Dolary a tečky

LISP/Python:

$$wordList\ docs = sort\ (nub\ (concat\ (map\ words\ docs)))$$

Trochu lepší:

$$wordList\ docs = sort\ \$\ nub\ \$\ concat\ \$\ map\ words\ docs$$

Ještě lepší:

$$wordList\ docs = sort \cdot nub \cdot concat \cdot map\ words\ \$\ docs$$

Nejlepší:

$$wordList = sort \cdot nub \cdot concat \cdot map\ words$$

Jak na IO

I/O není vedlejší efekt!

Místo toho máme speciální konstrukce, které dovolují hodnoty typu *IO a* kombinovat a vyrábět z nich větší *IO* hodnoty.

```
main =
  do putStr "Zadej cislo: "
     a ← getLine
     putStrLn "Vypis cisel od nuly:"
     printNums 0 (read a)

printNums :: Int → Int → IO ()
printNums a b
  | a ⩾ b     = return ()
  | otherwise = do print a
                   printNums (a + 1) b
```

$$
\begin{aligned}
&\mathit{group}\ [\,] = [\,] \\
&\mathit{group}\ l@(x\colon \_) = \mathit{takeWhile}\ (\equiv x)\ l \\
&\qquad\qquad\qquad\qquad : \mathit{group}\ (\mathit{dropWhile}\ (\equiv x)\ l) \\
&\mathit{qsort}\ [\,] \qquad = [\,] \\
&\mathit{qsort}\ (x : xs) = \mathit{qsort}\ a +\!\!+ x : \mathit{qsort}\ b \\
&\quad \textbf{where}\ a = \mathit{filter}\ (<x)\ xs \\
&\qquad\qquad b = \mathit{filter}\ (\geqslant x)\ xs \\
&\mathit{rle} = \mathit{map}\ (\lambda l \to (\mathit{head}\ l, \mathit{length}\ l)) \cdot \mathit{group}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{group}\ [\,] = [\,] \\
&\mathit{group}\ l@(x\colon \_) = \mathit{takeWhile}\ (\equiv x)\ l \\
&\qquad\qquad\qquad\quad : \mathit{group}\ (\mathit{dropWhile}\ (\equiv x)\ l) \\
&\mathit{qsort}\ [\,] \qquad = [\,] \\
&\mathit{qsort}\ (x : xs) = \mathit{qsort}\ a +\!\!+ x : \mathit{qsort}\ b \\
&\quad \textbf{where}\ a = \mathit{filter}\ (<x)\ xs \\
&\qquad\qquad b = \mathit{filter}\ (\geqslant x)\ xs \\
&\mathit{rle} = \mathit{map}\ (\lambda l \to (\mathit{head}\ l, \mathit{length}\ l)) \cdot \mathit{group}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{group} :: \mathit{Eq}\ a \Rightarrow [a] \to [[a]] \\
&\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a] \\
&\mathit{rle} :: \mathit{Eq}\ a \Rightarrow [a] \to [(a, \mathit{Int})]
\end{aligned}
$$

Praktikum — laziness

$$\mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})$$

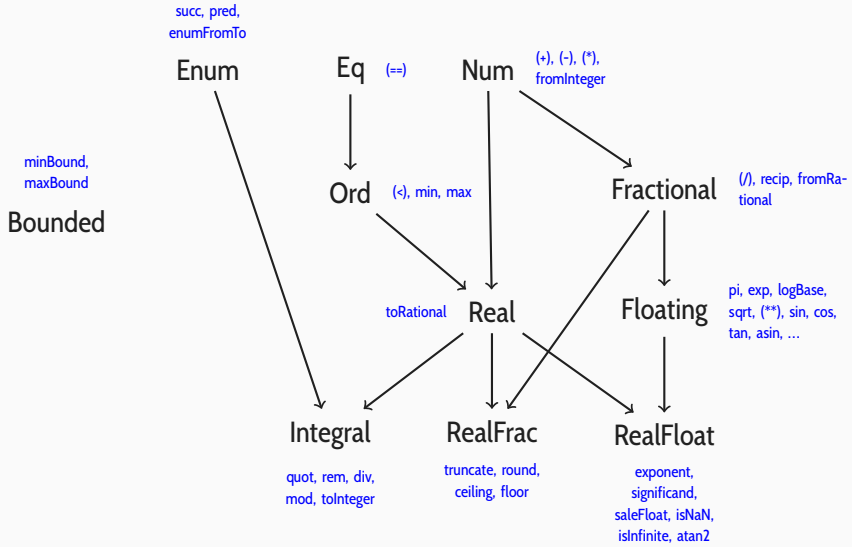
$$\mathit{findMin} = \mathit{head} \cdot \mathit{qsort}$$

Jakou má časovou složitost *findMin*?

Jednodušší typové třídy

třída	operace
Eq	$(\equiv), (\not\equiv)$
Ord	$(<), (\leqslant)$, *compare*, …
Enum	[1..10], [False..]
Num	$(+), (-), (*)$, *abs*, …
Integral	*div*, *mod*, …
Fractional	$(/)$, …
Floating	π, *sin*, *cos*, *exp*, *log*, *logBase*, *sqrt*, …
Show	*show*
Read	*read*

Číselná hierarchie



Číselná hierarchie vs. počítače

toRational 0.1

$\leadsto 3602879701896397 \,\%\, 36028797018963968$

ceiling $(1.0 \,/\, 0.0)$

$179769313486231590772930519078902473361 79 \ldots$

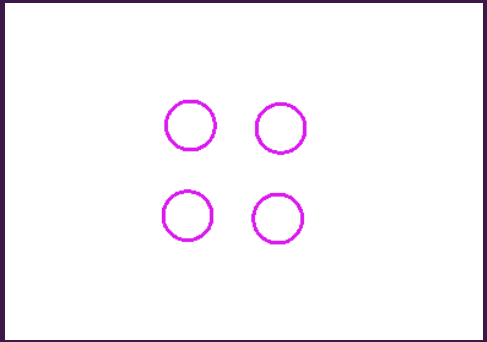
$(0.1 + 0.2 - 0.3) \,/\, (0.2 - 0.3 + 0.1)$

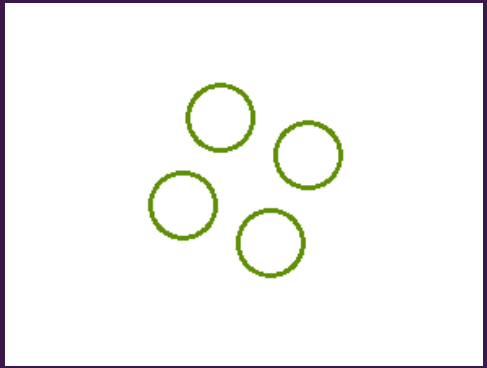
2.0

Moc teoretické?

Get something cool on screen in 10 minutes!

```
import Graphics.Gloss

main = animate FullScreen white $ λt →
  let blink ph sp = (1 + sin (ph + t * sp)) / 2
  in Pictures $ flip map [1 .. 4] $ λx →
      Rotate (90 * x + 20 * t) $
      Translate (80 + 40 * cos t) 0 $
      Color (makeColor (blink 0 1)
                       (blink 3 1)
                       (blink 0 π)
                       1) $
      ThickCircle 20 3
```





Overloading

Typové třídy existují kvůli přetěžování funkcí.

C-style přetěžování (“ad-hoc overloading”) je možné jen s jednosměrným typovým systémem, v Haskellu by bylo NP-úplné.

class *Moje a* **where**
  $moje :: a \to a \to a$

instance *Moje Int* **where**
  $moje = (+)$

instance *Moje String* **where**
  $moje = (+\!\!+)$

$moje\ 1\ 3 \quad \rightsquigarrow 4$

moje `"ah" "oj"` $\rightsquigarrow$ `"ahoj"`

Overloading — příklad

```
newtype I7 = I7 Int
instance Num I7 where
  (+) = i7lift2 (+)
  (-) = i7lift2 (-)
  (*) = i7lift2 (*)
  negate = i7lift negate
  abs = id
  signum (I7 0) = I7 0
  signum _ = I7 1
  fromInteger i = I7 $ i `mod` 7
i7lift f (I7 a) = I7 (f a `mod` 7)
i7lift2 f (I7 a) (I7 b) = I7 (f a b `mod` 7)
```

Overloading — příklad 2

```
data Infinite a = MinusInf | Finite a | PlusInf

instance Ord a => Ord (Infinite a) where
  compare (Finite a) (Finite b) = compare a b
  compare MinusInf  MinusInf    = EQ
  compare MinusInf  _           = LT
  compare _         MinusInf    = GT
  compare PlusInf   PlusInf     = EQ
  compare PlusInf   _           = GT
  compare _         PlusInf     = LT
```

Overloading — příklad 3

instance *Num a* $\Rightarrow$ *Num* $[a]$ **where**

$(+) = \mathit{zipWith}\ (+)$

$(-) = \mathit{zipWith}\ (-)$

$(*) = \mathit{zipWith}\ (*)$

$\mathit{negate} = \mathit{map}\ \mathit{negate}$

$\mathit{abs} = \mathit{map}\ \mathit{abs}$

$\mathit{signum} = \mathit{map}\ \mathit{signum}$

$\mathit{fromInteger} = \mathit{repeat} \cdot \mathit{fromInteger}$

$3 * [1,2,3] + 2 \quad \leadsto [5,8,11]$

$\mathit{fibs} = 0 : 1 : \mathit{fibs} + \mathit{tail}\ \mathit{fibs}$

Typové třídy nejsou interface!

OOP: `INumeric doSomething(INumeric a)`

vs.

Haskell: *doSomething* :: *Num a* $\Rightarrow$ *a* $\rightarrow$ *a*

Kde je rozdíl?

Typové třídy nejsou subtyping!

C-like: `(long)(double)(int)(float)5` je OK, integery jdou automaticky konvertovat.

Haskell: $((5 :: \mathit{Float}) :: \mathit{Int})$ je chyba!

Jak se tedy integer 5 zkonvertuje na float?!

Typové třídy nejsou subtyping!

C-like: `(long)(double)(int)(float)5` je OK, integery jdou automaticky konvertovat.

Haskell: $((5 :: \mathit{Float}) :: \mathit{Int})$ je chyba!

Jak se tedy integer 5 zkonvertuje na float?!

Desugaring literálů: pětka se přepíše na *fromInteger* 5, kde 5 je pevná *Integer*ová hodnota.

fromInteger :: *Num a* $\Rightarrow$ *Integer* $\rightarrow$ *a*

Důsledek: $5 :: \mathit{Num}\ a \Rightarrow a$

(konverze se většinou inlinuje a nemá runtime overhead)

Pologrupy a monoidy

class *Semigroup a* **where**
$(\diamond) :: a \to a \to a$

class *Semigroup a* $\Rightarrow$ *Monoid a* **where**
mempty $:: a$

Obecnější operace asociativního slepování (píše se jako <>):

$[1,2] \diamond [4,5] \quad \leadsto [1,2,4,5]$
`"ah" ◇ "oj"` $\leadsto$ `"ahoj"` -- i pro Text a ByteString
Just $[1,2] \diamond$ *Nothing* $\diamond$ *Just* $[3] \quad \leadsto$ *Just* $[1,2,3]$
EQ $\diamond$ *LT* $\diamond$ *GT* $\quad \leadsto$ *LT*
mempty $::$ *Maybe a* $\quad \leadsto$ *Nothing*

Nad množinou čísel existuje víc monoidů:

Sum $5 \diamond$ *Sum* $3 \quad \leadsto$ *Sum* 8
Product $5 \diamond$ *Product* $3 \quad \leadsto$ *Product* 15
(*mempty* $::$ *Product Int*) $\quad \leadsto$ *Product* 1

Kontejnerovitost

data *Tree a* $=$ *Nil* | *Branch* (*Tree a*) *a* (*Tree a*)

map $(+1)$ $[1, 2, 3]$ $\rightsquigarrow [2, 3, 4]$

map $(+1)$ (*Branch Nil* 1 (*Branch Nil* 2 *Nil*)) -- error

Co takhle přetížit map?

Kontejnerovitost

data *Tree a* = *Nil* | *Branch* (*Tree a*) *a* (*Tree a*)
map (+1) [1, 2, 3] ⇝ [2, 3, 4]
map (+1) (*Branch Nil* 1 (*Branch Nil* 2 *Nil*)) -- error

Co takhle přetížit map?

class *Functor f* **where** *fmap* :: (*a* → *b*) → (*f a* → *f b*)

Pozor, *f* má kind ∗ → ∗.

instance *Functor Tree* **where**
fmap _ *Nil* = *Nil*
fmap f (*Branch l a r*) = *Branch* (*fmap f l*) (*f a*) (*fmap f r*)
fmap (+1) [1, 2, 3] ⇝ [2, 3, 4]
fmap (+1) (*Branch Nil* 1 (*Branch Nil* 2 *Nil*)) ⇝ *Branch Nil* 2 (*Branch Nil* 3 *Nil*)

Kontejnerovitost

$(\langle\$\rangle) = \mathit{fmap}$

$(*2)\ \langle\$\rangle\ [2,3,4] \quad \rightsquigarrow [4,6,8]$

$(+1) \cdot (+2)\ \langle\$\rangle\ \mathit{Just}\ 5 \quad \rightsquigarrow \mathit{Just}\ 8$

$\mathit{uncurry}\ \mathit{gcd}\ \langle\$\rangle\ \mathit{Just}\ (1024, 768) \quad \rightsquigarrow \mathit{Just}\ 256$

$\mathit{fmap}\ \mathit{show}\ \langle\$\rangle\ [\mathit{Nothing}, \mathit{Just}\ 3, \mathit{Just}\ 4]$
$\quad \rightsquigarrow [\mathit{Nothing}, \mathit{Just}\ \texttt{"3"}, \mathit{Just}\ \texttt{"4"}]$


Vorsicht
Funktor
2 m Abstand halten

Rozumné funktory

$$fmap\ id \equiv id$$

$$fmap\ f \cdot fmap\ g \equiv fmap\ (f \cdot g)$$

Funkce jako data

Funkce jsou kontejnery na výsledky.

instance *Semigroup b* $\Rightarrow$ *Semigroup* $(a \to b)$ **where**
  $f \diamond g = \lambda a \to f\ a \diamond g\ a$
instance *Monoid b* $\Rightarrow$ *Monoid* $(a \to b)$ **where**
  $mempty = \lambda\_ \to mempty$

$(drop\ 3 \diamond take\ 3)\ [1,2,3,4,5] \quad \rightsquigarrow [4,5,1,2,3]$
$(drop \diamond take)\ 2\ [1,2,3,4,5] \quad \rightsquigarrow [3,4,5,1,2]$

Podobně:

instance *Functor* $((\to)\ p)$ **where**
  $fmap = (\cdot)$

$fmap :: (a \to b) \to ((p \to a) \to (p \to b))$
$(show \langle\$\rangle\ (*3)) :: Int \to String$
$(show \langle\$\rangle\ (*3))\ 3 \quad \rightsquigarrow$ `"9"`

Funkce jako data

instance *Num b* $\Rightarrow$ *Num* $(a \to b)$ **where**

$$
\begin{aligned}
(f + g)\ x &= f\ x + g\ x \\
(f * g)\ x &= f\ x * g\ x \\
\mathit{negate}\ f\ x &= \mathit{negate}\ (f\ x) \\
\mathit{fromInteger}\ n\ x &= \mathit{fromInteger}\ n \\
\mathit{signum}\ f\ x &= \mathit{signum}\ (f\ x) \\
\mathit{abs}\ f\ x &= \mathit{abs}\ (f\ x)
\end{aligned}
$$

$$
\begin{aligned}
(\mathit{sin}\hat{}2 + \mathit{cos}\hat{}2)\ 4.64234 &\rightsquigarrow 1.0 \\
(\mathit{sin} + 1)\ 1 &\rightsquigarrow 1.8414709848078965 \\
(\mathit{sin} + 1)\ \pi &\rightsquigarrow 1.0000000000000002 \\
(\mathit{negate}\ \mathit{negate})\ 5 &\rightsquigarrow 5 \\
3\ 7 &\rightsquigarrow 3 \\
((+) * (-))\ 7\ 5 &\rightsquigarrow 24
\end{aligned}
$$

Funkce vs. kontejnery

$$
\begin{aligned}
(\$) &:: (a \to b) \to (\ a \to\ b\ ) \\
map &:: (a \to b) \to ([a] \to [b]) \\
fmap &:: (a \to b) \to (f\ a \to f\ b) \\
\_ &:: f\ (a \to b) \to (f\ a \to f\ b)
\end{aligned}
$$

Co když je původní funkce taky v kontejneru?

Funkce vs. kontejnery

Např.:

$$
\begin{aligned}
&\textbf{let } a = \mathit{getProgram} :: d \to r \\
&\quad\;\; b = \mathit{getData} :: d \\
&\textbf{in } a\ b
\end{aligned}
$$

Když získávání programu nebo dat může selhat, použijeme Maybe:

$$
\begin{aligned}
&\textbf{let } a = \mathit{getProgram} :: \mathit{Maybe}\ (d \to r) \\
&\quad\;\; b = \mathit{getData} :: \mathit{Maybe}\ d \\
&\textbf{in } a\ \text{`}\mathit{ap}\text{`}\ b \\
&\quad \textbf{where } \mathit{ap}\ (\mathit{Just}\ l)\ (\mathit{Just}\ r) = \mathit{Just}\ (l\ r) \\
&\qquad\quad\;\; \mathit{ap}\ \_\qquad\quad \_\qquad\quad = \mathit{Nothing}
\end{aligned}
$$

Applicative

Koncept aplikace “obalené” funkce je zachycený ve třídě *Applicative*:

class *Functor f* $\Rightarrow$ *Applicative f* **where**
  pure :: $a \to f\ a$
  $(\langle * \rangle) :: f\ (a \to b) \to f\ a \to f\ b$

Just show	$\langle * \rangle$ *Just* 3	$\rightsquigarrow$ *Just* `"3"`
(*pure show* $\langle * \rangle$ *pure* 3) :: *Maybe String*		$\rightsquigarrow$ *Just* `"3"`
Nothing	$\langle * \rangle$ *Just* 3	$\rightsquigarrow$ *Nothing*
Just show	$\langle * \rangle$ *Nothing*	$\rightsquigarrow$ *Nothing*
Just gcd	$\langle * \rangle$ *Just* 1024 $\langle * \rangle$ *Just* 768	$\rightsquigarrow$ *Just* 256
(*pure show* $\langle * \rangle$ *pure* 3) :: [*String*]		$\rightsquigarrow$ `["3"]`

Instance Applicative

instance	sémantika
Maybe	selhávání
Either l	selhávání s chybou typu *l*
[ ]	množiny výsledků výpočtu
$(\rightarrow)$ *p*	výpočet s globálním parametrem

Příklady Applicative

$$\mathit{Just}\ (,)\ \langle * \rangle\ \mathit{Just}\ 3\ \langle * \rangle\ \mathit{pure}\ 5 \quad \rightsquigarrow \mathit{Just}\ (3,5)$$
$$(,)\ \langle \$ \rangle\ \mathit{pure}\ 3\ \langle * \rangle\ \mathit{Just}\ 5 \quad \rightsquigarrow \mathit{Just}\ (3,5)$$

$$((+)\ \langle \$ \rangle\ \mathit{pure}\ 3\ \langle * \rangle\ \mathit{pure}\ 7) :: \mathit{Either\ String\ Int}$$
$$\rightsquigarrow \mathit{Right}\ 10$$

$$(+)\ \langle \$ \rangle\ \mathit{Right}\ 3\ \langle * \rangle\ \mathit{pure}\ 7$$
$$\rightsquigarrow \mathit{Right}\ 10$$

$$(+)\ \langle \$ \rangle\ \mathit{pure}\ 3\ \langle * \rangle\ \mathit{Left}\ \texttt{"sorry error"}$$
$$\rightsquigarrow \mathit{Left}\ \texttt{"sorry error"}$$

$$[(+1),(+2),\mathit{negate}]\ \langle * \rangle\ [2,3,4]$$
$$\rightsquigarrow [3,4,5,4,5,6,-2,-3,-4]$$

$$\mathit{rle} = \mathit{map}\ (\quad (,)\ \langle \$ \rangle\ \mathit{head}\ \langle * \rangle\ \mathit{length}) \cdot \mathit{group}$$
$$\mathit{rle}' = \mathit{map}\ (\mathit{liftA2}\ (,)\quad \mathit{head}\quad \mathit{length}) \cdot \mathit{group}$$

Rozumné aplikativní funktory

$$pure\ id\ \langle * \rangle\ a \equiv a$$

$$pure\ (a\ b) \equiv pure\ a\ \langle * \rangle\ pure\ b$$

$$a\ \langle * \rangle\ pure\ b \equiv pure\ (\$b)\ \langle * \rangle\ a$$

$$a\ \langle * \rangle\ (b\ \langle * \rangle\ c) \equiv pure\ (\cdot)\ \langle * \rangle\ a\ \langle * \rangle\ b\ \langle * \rangle\ c$$

Applicative vs. monoid

Užitečná demonstrativní instance:

$$\begin{aligned}
&\textbf{instance } \mathit{Monoid}\ a \Rightarrow \mathit{Applicative}\ ((,)\ a)\ \textbf{where}\\
&\quad \mathit{pure}\ x \qquad\quad = (\mathit{mempty}, x)\\
&\quad (u, f)\ \langle * \rangle\ (v, x) = (u \diamond v, \quad f\ x)
\end{aligned}$$

$$\begin{aligned}
&(\texttt{"hello "}, (+17))\ \langle * \rangle\ (\texttt{"world!"}, 2003)\\
&\qquad \rightsquigarrow (\texttt{"hello world!"}, 2020)
\end{aligned}$$

Funkce vs. kontejnery II.

$$
\begin{aligned}
(\$) &:: (a \to b) \to (a \to b) \\
\mathit{fmap} &:: (a \to b) \to (f\ a \to f\ b) \\
(\langle * \rangle) &:: f\ (a \to b) \to (f\ a \to f\ b) \\
\_ &:: (a \to f\ b) \to (f\ a \to f\ b)
\end{aligned}
$$

Jde funkci, jejíž výsledek je obalený nějakou sémantikou (např.: může selhat, nebo možných výsledků může být víc), předělat na funkci, která tento obal spojí s již existujícím obalem?

Podobný problém z jiné strany:

$$\mathit{join} :: f\ (f\ a) \to f\ a$$

Spojování Applicative

Příklad s *Maybe*: Nějaké funkce můžou selhat, potřebujeme z nich vyrobit jednu uniformně selhávající funkci.

$$
\begin{aligned}
&\mathit{lookup} \;\; :: \mathit{Eq}\ a \Rightarrow a \to [(a,b)] \to \mathit{Maybe}\ b \\
&\mathit{lookup4} :: \mathit{Eq}\ a \Rightarrow a \to [(a,a)] \to \mathit{Maybe}\ a \\
&\mathit{lookup4}\ a\ l = \textbf{case}\ \mathit{lookup}\ a\ l\ \textbf{of} \\
&\quad \mathit{Nothing} \to \mathit{Nothing} \\
&\quad \mathit{Just}\ a' \to \textbf{case}\ \mathit{lookup}\ a'\ l\ \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad \mathit{Just}\ a'' \to \textbf{case}\ \mathit{lookup}\ a''\ l\ \textbf{of} \\
&\qquad\quad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad\quad \mathit{Just}\ a''' \to \mathit{lookup}\ a'''\ l
\end{aligned}
$$

SW Engineering: Vytrhněme kus opakujícího se kódu!

andThen

$$
\begin{aligned}
&\mathit{andThen} :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\
&\mathit{andThen}\ \mathit{Nothing}\ \_ = \mathit{Nothing} \\
&\mathit{andThen}\ (\mathit{Just}\ a)\ f = f\ a \\
&\mathit{lookup4}\ a\ l = \mathit{lookup}\ a\ l\ `\mathit{andThen}`\ \lambda a \to \\
&\qquad\qquad\qquad \mathit{lookup}\ a\ l\ `\mathit{andThen}`\ \lambda a \to \\
&\qquad\qquad\qquad \mathit{lookup}\ a\ l\ `\mathit{andThen}`\ \lambda a \to \\
&\qquad\qquad\qquad \mathit{lookup}\ a\ l
\end{aligned}
$$

Trochu lepší!

andThen (na steroidech)

```
infixr 1 `andThen`
lookup4 a l = go a `andThen` go
                   `andThen` go
                   `andThen` go
  where go a = lookup a l
```

Seznamy

Úkol: Robot za jednu časovou jednotku může jít do nějakých (na pozici závislých) směrů, tam může udělat nějaké (na pozice závislé) akce, z toho je hromada nových možností kam může dojít. Najděte je.

$$
\begin{aligned}
&\mathit{directionsAt} :: \mathit{Location} &&\to [\mathit{Direction}] \\
&\mathit{go} :: \mathit{Direction} \to \mathit{Location} &&\to \mathit{Location} \\
&\mathit{actionsAt} :: \mathit{Location} &&\to [\mathit{Action}] \\
&\mathit{act} :: \mathit{Action} \to \mathit{Location} &&\to \mathit{Location}
\end{aligned}
$$

```
directionsAt :: Location                → [Direction]
go           :: Direction → Location → Location
actionsAt    :: Location                → [Action]
act          :: Action → Location       → Location


possibleResults :: Location → [Location]
possibleResults loc = concat $
  map (λdir → let newloc = go dir loc
              in map (λaction → act action newloc)
                     (actionsAt newloc)
      )
      (directionsAt loc)
```

...hm.

withTheseCases

$$withTheseCases :: [a] \to (a \to [b]) \to [b]$$
$$withTheseCases\ a\ f = concat\ (map\ f\ a)$$

infixr 2 ‘*withTheseCases*‘

withTheseCases

```
withTheseCases :: [a] → (a → [b]) → [b]
withTheseCases a f = concat (map f a)
infixr 2 `withTheseCases`
```

```
possibleResults loc =
  directionsAt loc `withTheseCases` λdir →
  let newloc = go dir loc
  in actionsAt newloc `withTheseCases` λaction →
    [act action newloc]
```

(poslední řádek vrací jednu možnost)

Generické andThen

$$andThen :: Maybe\ a \to (a \to Maybe\ b) \to Maybe\ b$$
$$withTheseCases :: [a] \to (a \to [b]) \to [b]$$

Generické andThen

$$andThen :: Maybe\ a \to (a \to Maybe\ b) \to Maybe\ b$$
$$withTheseCases :: [a] \to (a \to [b]) \to [b]$$

$$\begin{array}{lll}
(\$) & :: & (a \to b) \to (a \to b) \\
fmap & :: Functor\ f \Rightarrow & (a \to b) \to (f\ a \to f\ b) \\
(\langle * \rangle) & :: Applicative\ f \Rightarrow f\ & (a \to b) \to (f\ a \to f\ b) \\
flip\ (\gg\!=) & :: Monad\ m \Rightarrow & (a \to m\ b) \to (m\ a \to m\ b)
\end{array}$$

Generické *andThen* a *withCases* se jmenuje $(\gg\!=)$, čte se “bind”.

Jméno: Aplikativní monoid (skoro).

$$\begin{aligned}&\textbf{class } \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where}\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\\ &\quad (\gg) :: m\ a \to m\ b \to m\ b\\ &\quad \mathit{return} :: a \to m\ a\end{aligned}$$

$(\gg)$ zahodí ‘výsledek’, zachová jen změny v ‘obalu’.

Monády alternativně

$$join :: Monad\ m \Rightarrow m\ (m\ a) \rightarrow m\ a$$

V rozumných případech platí:

$$(a \gg\!= f) \equiv join\ (f \langle\$\rangle\ a)$$

Pozn.: chování joinu pro seznamy a Maybe:

$$joinL :: [[a]] \rightarrow [a]$$
$$joinL = concat$$
$$joinMaybe :: Maybe\ (Maybe\ a) \rightarrow Maybe\ a$$
$$joinMaybe\ (Just\ x) = x$$
$$joinMaybe\ \_ = Nothing$$


MONADS!
NO! PLEASE, STOP!
THEY'RE STILL CHILDREN
@impurepics

Rozumné monády

$$return\ a \gg\!= f \equiv f\ a$$

$$a \gg\!= return \equiv a$$

$$a \gg\!= (\lambda x \to b\ x \gg\!= c) \equiv (a \gg\!= b) \gg\!= c$$

Imperativní syntax

Protože používat $\gg=$ a $\gg$ je otrava, Haskell má imperativní **do**-syntax.

$$
\begin{aligned}
fn = \textbf{do}\ & a \leftarrow f \\
& b \leftarrow g \\
& h\ a\ b \\
& h\ b\ a
\end{aligned}
$$

...se přepíše na:

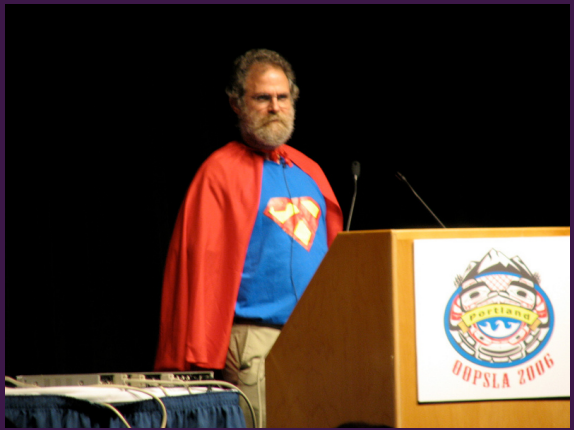
$$
\begin{aligned}
fn = f \gg= \lambda a \rightarrow & \\
g \gg= \lambda b \rightarrow & \\
h\ a\ b \gg & \\
h\ b\ a &
\end{aligned}
$$

Imperativní syntax

```
lookup4 a l =
  do a ← lookup a l
     a ← lookup a l
     a ← lookup a l
     a ← lookup a l
     return a

possibleResults loc =
  do dir ← directionsAt loc
     let newloc = go dir loc
     action ← actionsAt newloc
     return $ act action newloc
```

Philip Lee Wadler

Philip Lee Wadler



The Lambdaman

List comprehension

Speciální syntaxe pro seznamy:

$$
\begin{aligned}
&\mathit{possibleResults}\ \mathit{loc} = \\
&\quad [\mathit{act}\ \mathit{action}\ \mathit{newloc} \mid \mathit{dir} \leftarrow \mathit{directionsAt}\ \mathit{loc}, \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ \mathit{newloc} = \mathit{go}\ \mathit{dir}\ \mathit{loc}, \\
&\qquad\qquad\qquad\qquad\ \ \mathit{action} \leftarrow \mathit{actionsAt}\ \mathit{newloc}]
\end{aligned}
$$

Jednodušší použití:

$$[(x, y) \mid x \leftarrow [1\,..], y \leftarrow [1\,..x], x\ `\mathit{mod}`\ y \equiv 0]$$

$$\mathit{map'}\ f\ l = [f\ a \mid a \leftarrow l]$$

$$\mathit{filter'}\ f\ l = [a \mid a \leftarrow l, f\ a]$$

(“podmínky” se automaticky obalí pomocí *guard*)

Standardní definice Monad Maybe

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing = Nothing
instance Applicative Maybe where
  pure = Just
  Just f <*> Just a = Just (f a)
  _ <*> _ = Nothing
instance Monad Maybe where
  return = pure
  Just a >>= f = f a
  _ >>= _ = Nothing
```

Standardní definice Monad []

```
instance Functor [] where
  fmap = map
instance Applicative [] where
  pure x = [x]
  fs <*> as = concatMap (flip map as) fs
instance Monad [] where
  return = pure
  as >>= f = concatMap f as
```

Extra příklad: DebugLog

Zapišme si postup výpočtu do logu k hodnotě.

```
data Val a = Val a String deriving Show

liftVal1 op str (Val a logA) = Val (op a)
                                   (str ++ "(" ++ logA ++ ")")
liftVal2 op str (Val a logA) (Val b logB) =
  Val (a `op` b) $ concat ["(", logA, ")", str, "(", logB, ")"]

instance (Num a, Show a) => Num (Val a) where
  (+) = liftVal2 (+) "+"
  (-) = liftVal2 (-) "-"
  (*) = liftVal2 (*) "*"
  negate = liftVal1 negate "negate"
  abs = liftVal1 abs "abs"
  fromInteger x = Val (fromInteger x) (show x)
```

Extra příklad: DebugLog

$$(1 + \mathit{abs}\ (2 * \mathit{negate}\ 3)) :: \mathit{Val}\ \mathit{Int}$$
$$\rightsquigarrow \mathit{Val}\ 7\ \texttt{"(1)+(abs((2)*(negate(3))))"}$$

Celý výpočet ale většinou není na jeden řádek...

Extra příklad: DebugLog

```
instance Functor Val where
  fmap f (Val a log) = Val (f a) log
instance Applicative Val where
  pure a = Val a "?"
  Val f logF <*> Val a logA =
    Val (f a) $ concat ["(", logF, ") (", logA, ")"]
instance Monad Val where
  return = pure
  Val a logA >>= f =
    let Val b logB = f a
    in Val b (logA ++ ";\n" ++ logB)
mkVal a = Val a (show a)
nameVal n (Val a log) = Val (Val a n) $ n ++ "=" ++ log
```

Extra příklad: DebugLog

$$
\begin{aligned}
&\mathit{solve}\ a'\ b'\ c' = \mathbf{do}\\
&\quad \mathbf{let}\ a = \mathit{mkVal}\ a'\\
&\qquad\ \ b = \mathit{mkVal}\ b'\\
&\qquad\ \ c = \mathit{mkVal}\ c'\\
&\quad d \leftarrow \mathit{nameVal}\ \texttt{"D"}\ \$\ b\texttt{\^{}}2 - 4 * a * c\\
&\quad (-b + \mathit{sqrt}\ d)\ /\ (2 * a)
\end{aligned}
$$

$\mathit{solve}\ 5\ 3\ (-10)$

$\rightsquigarrow$ *Val* 1.145683229480096 `"D=((3.0)*(3.0))-(((4)*(5.0))*(-10.0));((negate(3.0))+(odmocnina(D)))/((2)*(5.0))"`

(instanci pro *Fractional* a *Floating* někdo doplnil)

IO Flashback

```
putStrLn :: String → IO ()
getLine  :: IO String

main :: IO ()
main = do putStrLn "dej jmeno!"
          jmeno ← getLine
          putStrLn $
            "no nazdar je tady " ++ jmeno
```

Jak se vyrobí IO?

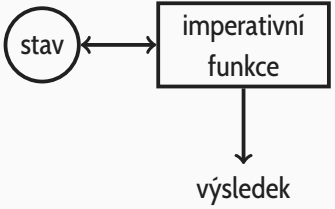
Myšlenka: IO akce musí přece manipulovat s nějakým globálním stavem! Vyrobíme monádu, která ukládá stav, na ní pak půjde stavět dál.

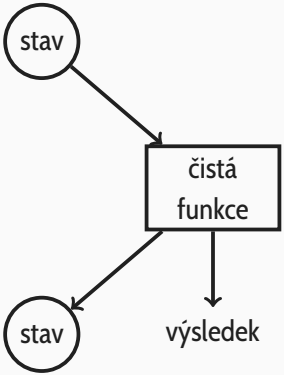
Požadavky:

- $put :: s \to m\ ()$
- $get :: m\ s$
- tohle vrátí *True*:

$$\begin{aligned}&\textbf{do}\ put\ a\\&\quad b \leftarrow get\\&\quad return\ \$\ a \equiv b\end{aligned}$$

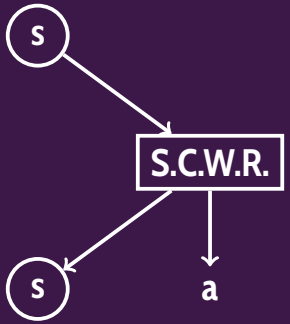
Změna stavu bez vedlejšího efektu





State

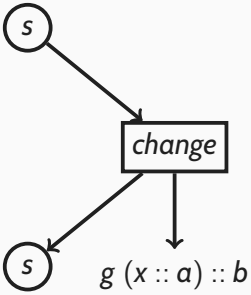
$$\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{StateChangeWithResult}\ (s \to (s, a))$$



Stavový výpočet je kontejner!

instance *Functor* (*State s*) **where**

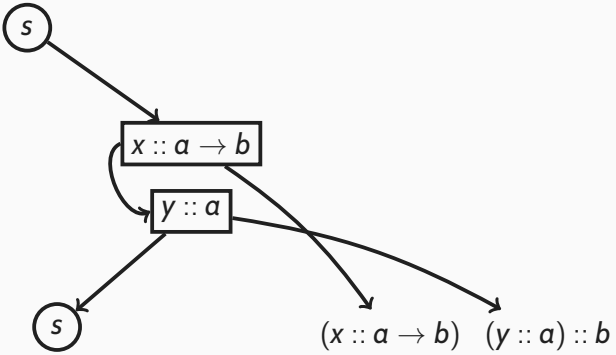
fmap g change =



Stavový výpočet je aplikativní!

instance *Applicative* (*State s*) **where**

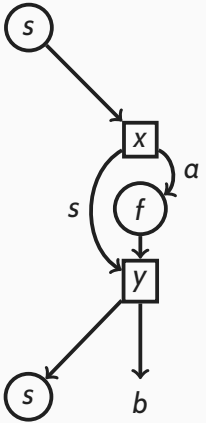
$x \langle * \rangle y =$



Stavový výpočet je monáda!

instance *Monad* (*State s*) **where**

$x \gg\!= f =$



Implementace State

```
newtype State s a = StateChange (s -> (s, a))

instance Functor (State s) where
  fmap func (StateChange f) =
    StateChange (fmap func . f)

instance Applicative (State s) where
  pure a = StateChange (\s -> (s, a))
  StateChange f <*> StateChange v =
    StateChange (\s -> let (s1, a1) = f s
                           (s2, a2) = v s1
                       in (s2, a1 a2))
```

Implementace State

```
instance Monad (State s) where
  return = pure
  StateChange f1 >>= f2 =
    StateChange (λs → let (s', a) = f1 s
                          StateChange f3 = f2 a
                      in f3 s')
```

Manipulace stavu

$$put :: s \to State\ s\ ()$$
$$put\ ns = StateChange\ (\lambda\_ \to (ns, ()))$$

$$get :: State\ s\ s$$
$$get = StateChange\ (\lambda s \to (s, s))$$

$$modify :: (s \to s) \to State\ s\ ()$$
$$modify\ f = StateChange\ (\lambda s \to (f\ s, ()))$$

$$execState\ (StateChange\ f)\ s = snd\ \$\ f\ s$$

Použití State

```
fact n = execState (factS n) 1

factS n
  | n ⩽ 1 = do result ← get
               return result
  | otherwise = do modify (∗n)
                   factS (n − 1)

getRand :: Int → State Int Int
getRand max =
  do newSeed ← (12345+) · (1103515245∗) ⟨$⟩ get
     put newSeed
     return $ (newSeed `div` 65536) `mod` max
```

(viz. `man 3 rand`)

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{get2BoringRand} :: \mathit{Int} \to \mathit{State}\ \mathit{Int}\ (\mathit{Int}, \mathit{Int}) \\
&\mathit{get2BoringRand}\ \mathit{max} = \mathbf{do} \\
&\quad a \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad b \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad \mathit{return}\ (a, b)
\end{aligned}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{get2BoringRand} :: \mathit{Int} \to \mathit{State}\ \mathit{Int}\ (\mathit{Int}, \mathit{Int}) \\
&\mathit{get2BoringRand}\ \mathit{max} = \mathbf{do} \\
&\quad a \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad b \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad \mathit{return}\ (a, b)
\end{aligned}
$$

$$
\mathit{get2Rand}\ \mathit{max} = \mathit{pure}\ (,)\ \langle * \rangle\ \mathit{getRand}\ \mathit{max}\ \langle * \rangle\ \mathit{getRand}\ \mathit{max}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{get2BoringRand} :: \mathit{Int} \to \mathit{State}\ \mathit{Int}\ (\mathit{Int}, \mathit{Int}) \\
&\mathit{get2BoringRand}\ \mathit{max} = \mathbf{do} \\
&\quad a \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad b \leftarrow \mathit{getRand}\ \mathit{max} \\
&\quad \mathit{return}\ (a, b)
\end{aligned}
$$

$$
\mathit{get2Rand}\ \mathit{max} = \mathit{pure}\ (,)\ \langle * \rangle\ \mathit{getRand}\ \mathit{max}\ \langle * \rangle\ \mathit{getRand}\ \mathit{max}
$$

$$
\mathit{get2Rand}'\ \mathit{max} = (,)\ \langle \$ \rangle\ \mathit{getRand}\ \mathit{max}\ \langle * \rangle\ \mathit{getRand}\ \mathit{max}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{getNRand}\ \mathit{max}\ 0 = \mathit{pure}\ [\,] \\
&\mathit{getNRand}\ \mathit{max}\ n = (:)\ \langle\$\rangle\ \mathit{getRand}\ \mathit{max} \\
&\qquad\qquad\qquad\qquad \langle*\rangle\ \mathit{getNRand}\ \mathit{max}\ (n-1)
\end{aligned}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{getNRand}\ \mathit{max}\ 0 = \mathit{pure}\ [\,] \\
&\mathit{getNRand}\ \mathit{max}\ n = (:)\ \langle\$\rangle\ \mathit{getRand}\ \mathit{max} \\
&\qquad\qquad\qquad\qquad \langle * \rangle\ \mathit{getNRand}\ \mathit{max}\ (n - 1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNRand'}\ \mathit{max}\ n = \mathit{mapM}\ (\lambda\_ \to \mathit{getRand}\ \mathit{max})\ [1\,..\,n] \\
&\mathit{getNRand''}\ \mathit{max}\ n = \mathit{traverse}\ (\mathit{const}\ \$\ \mathit{getRand}\ \mathit{max})\ [1\,..\,n]
\end{aligned}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{getNRand}\ \mathit{max}\ 0 = \mathit{pure}\ [\,] \\
&\mathit{getNRand}\ \mathit{max}\ n = (:)\ \langle\$\rangle\ \mathit{getRand}\ \mathit{max} \\
&\qquad\qquad\qquad\qquad \langle*\rangle\ \mathit{getNRand}\ \mathit{max}\ (n-1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNRand'}\ \mathit{max}\ n = \mathit{mapM}\ (\lambda\_ \to \mathit{getRand}\ \mathit{max})\ [1\,..\,n] \\
&\mathit{getNRand''}\ \mathit{max}\ n = \mathit{traverse}\ (\mathit{const}\ \$\ \mathit{getRand}\ \mathit{max})\ [1\,..\,n]
\end{aligned}
$$

$$
\mathit{getNRand'''}\ \mathit{max}\ n = \mathit{replicateM}\ n\ \$\ \mathit{getRand}\ \mathit{max}
$$

Použití State jako Applicative

$$
\begin{aligned}
&\mathit{getNRand}\ \mathit{max}\ 0 = \mathit{pure}\ [\,] \\
&\mathit{getNRand}\ \mathit{max}\ n = (:)\ \langle\$\rangle\ \mathit{getRand}\ \mathit{max} \\
&\qquad\qquad\qquad\qquad \langle*\rangle\ \mathit{getNRand}\ \mathit{max}\ (n-1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNRand'}\ \mathit{max}\ n = \mathit{mapM}\ (\lambda\_ \to \mathit{getRand}\ \mathit{max})\ [1\,..\,n] \\
&\mathit{getNRand''}\ \mathit{max}\ n = \mathit{traverse}\ (\mathit{const}\ \$\ \mathit{getRand}\ \mathit{max})\ [1\,..\,n]
\end{aligned}
$$

$$
\mathit{getNRand'''}\ \mathit{max}\ n = \mathit{replicateM}\ n\ \$\ \mathit{getRand}\ \mathit{max}
$$

$$
\mathit{getNRand''''} = \mathit{flip}\ \mathit{replicateM} \cdot \mathit{getRand}
$$

Pravda o IO

```
$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/
Prelude> :i IO
newtype IO a
  = GHC.Types.IO
      (GHC.Prim.State# GHC.Prim.RealWorld
         -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
```

Kolik paměti spotřebuje uložení *RealWorld*?

Kolik paměti spotřebuje uložení *RealWorld*?

Je to jen “token” typu *Void*.

A co že je teda monáda?

A co že je teda monáda?

Praxe s monádami: Parsovací kombinátory

Jak funguje obyčejný left-to-right rekurzivní parser?

- postupně vytváří parsovanou strukturu (lazy!)
- pamatuje si místo, kde přestal parsovat (“zbytek”)

Idea parsovacích kombinátorů:

$$\textbf{newtype}\ \mathit{Parser}\ r = \mathit{Parser}\ (\mathit{State}\ \mathit{Pozice}\ r)$$


Vorsicht
Funktor
2 m Abstand halten

Parsec demo

Kam se chceme dostat:

```
parenthesized x = do char '('
                     res ← x
                     char ')'
                     return res

numberOrWordOrBoth = number ⟨|⟩ word ⟨|⟩ (number ≫ word)

parseNum :: (Num a, Read a) ⇒ Parser a
parseNum = (read ⟨$⟩ many1 parseDigit)
           ⟨|⟩ do char '-'
                  negate · read ⟨$⟩ many1 parseDigit
```

Alternative

Alternative obsahuje ‘prázdný’ prvek a operaci na slepování výpočtů (podobně jako *Monoid* slepuje hodnoty).

Sémantika: Pokud může výpočet dopadnout více možnostmi, *Alternative* umožňuje specifikovat další možnosti. Pokud některá větev výpočtu selže, použije se jiná.

class *Applicative f* $\Rightarrow$ *Alternative f* **where**
  empty :: *f a*
  $(\langle|\rangle)$ :: *f a* $\rightarrow$ *f a* $\rightarrow$ *f a*

Užitečné instance: *Maybe*, [ ], *Validation*, *Except*, …, *Parser r*

Implementace jednoduchého parseru

```
newtype Parser a = Parser (State String (Maybe a))

instance Functor Parser where
  fmap f (Parser s) = Parser $ fmap (fmap f) s

instance Applicative Parser where
  pure a = Parser . pure . pure $ a
  Parser l <*> Parser r = Parser $ do
    a <- l
    case a of
      Nothing -> pure Nothing
      Just f -> fmap (fmap f) r
```

Implementace jednoduchého parseru

```
instance Monad Parser where
  return = pure
  Parser l >>= p =
    Parser $ do a <- l
                case a of
                  Nothing -> return Nothing
                  Just a -> let Parser r = p a in r
```

Implementace jednoduchého parseru

```
instance Alternative Parser where
  empty = Parser $ pure empty
  Parser l <|> Parser r =
    Parser $ do backup <- get
                l' <- l
                case l' of
                  Nothing -> put backup >> r
                  a -> return a
```

Pomůcky

```
parseFail = empty
parseGet = Parser $ Just <$> get
parseSet s = Parser $ Just <$> put s

pEof = do s <- parseGet
          case s of
            "" -> return ()
            _ -> parseFail

doParse (Parser p) = runState p
doParseEof p = fst . doParse (p >>= \r -> pEof >> return r)
```

Pomůcky

$$
\begin{aligned}
&\mathit{pAny} = \textbf{do}\ s \leftarrow \mathit{parseGet} \\
&\qquad\qquad \textbf{case}\ s\ \textbf{of} \\
&\qquad\qquad\quad (c : cs) \rightarrow \mathit{parseSet}\ cs \gg \mathit{return}\ c \\
&\qquad\qquad\quad \_ \rightarrow \mathit{parseFail} \\
&\mathit{pCharCond}\ f = \mathit{pAny} \gg\!= \lambda c \rightarrow \textbf{if}\ f\ c\ \textbf{then}\ \mathit{return}\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{parseFail} \\
&\mathit{pOneOf}\ cs = \mathit{pCharCond}\ (\in cs) \\
&\mathit{pAnyExcept}\ cs = \mathit{pCharCond}\ \$\ \neg \cdot (\in cs) \\
&\mathit{pChar}\ c = \mathit{pCharCond}\ (\equiv c) \\
&\mathit{pStr}\ s = \mathit{mapM}\ \mathit{pChar}\ s
\end{aligned}
$$

Kombinátory

```
pMany1 p = do x ← p
              xs ← pMany p
              return (x : xs)

pMany p = pMany1 p ⟨|⟩ pure []

pBracketed l r p = do l
                      res ← p
                      r
                      return res

pDelim l r = pBracketed (pStr l) (pStr r)
pBrackets = pDelim "[" "]"
pBraces = pDelim "{" "}"

pQuoted q = pDelim q q · pMany $ pAnyExcept q
```

Sekvence s oddělovačem

```
infixr 4 <:>
a <:> b = (:) <$> a <*> b

pSep1 sep p = p <:> (sep >> pSep1 sep p)
                <|>
                (:[]) <$> p    -- also: fmap pure p

pSep sep p = pSep1 sep p <|> pure []

pCommaDelimited = pSep (pChar ',')
```

Parser JSONu!

```
data JSON = JSInt Int
          | JSStr String
          | JSList [JSON]
          | JSObj [(String, JSON)]

pJSON = pJSInt <|> pJSStr <|> pJSList <|> pJSObj

pJSInt = JSInt . (read :: String -> Int)
           <$> pMany1 (pOneOf ['0'..'9'])
pJSStr = JSStr <$> (pQuoted "\"" <|> pQuoted "'")

pJSList = JSList <$> pBrackets (pCommaDelimited pJSON)

pJSObj = JSObj <$> pBraces (pCommaDelimited objItem)
  where objItem = do JSStr key <- pJSStr
                     pChar ':'
                     ((,) key) <$> pJSON
```

Jak na whitespace?

```
import Data.Char

whitespace = pMany $ pCharCond isSpace
lexeme = (whitespace >>)
```

Dostupné knihovny

Balíček `parsec` je původní implementace, obsahuje většinu zmíněných kombinátorů.

```
import Text.Parsec.String (Parser)
import Text.Parsec.Char
import Text.Parsec (try)
import Text.Parsec.Combinator (between, eof, many)
```

try se používá pro backtrackování:

- v některých případech backtrackovat není vhodné (např. při zotavení z chyb)
- zbytečné referencování pozice není úplně úsporné

```
try (char 'a') <|> char 'b'
```

Dostupné knihovny

Megaparsec — novější varianta Parsecu:

- lepší chybové hlášky (unexpected vs. expected, kumulované chyby)
- rychlejší zpracování
- hromada utilit
- parsování *Stream*ů

Attoparsec — miniaturní varianta

- méně featur
- víc rychlosti díky specializaci na *ByteString*y
  (rychlost je porovnatelná s ručně optimalizovanými parsery v C)

Balíčky a knihovny

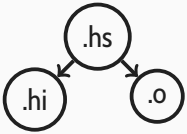
Oddělený překlad

Překládat programy celé najednou je poměrně náročné

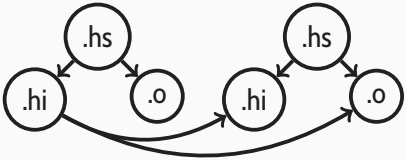
- něco to trvá
- spotřebuje to hodně paměti
- exploze složitosti při optimalizaci

Tradiční řešení: oddělený překlad po modulech

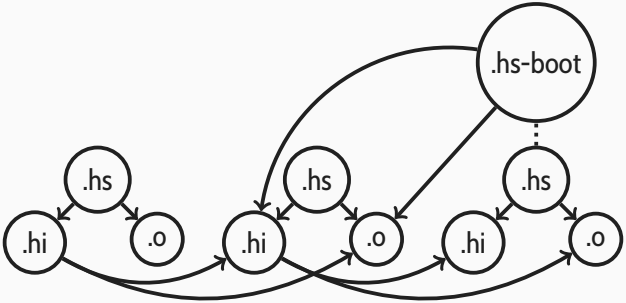
Oddělený překlad



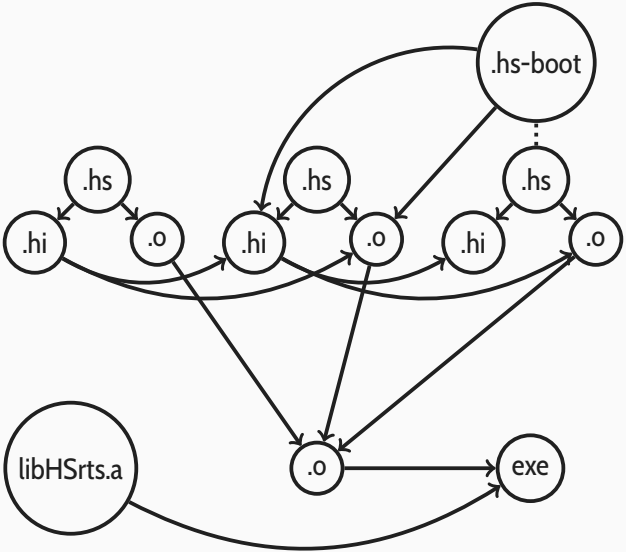
Oddělený překlad



Oddělený překlad



Oddělený překlad



Jak importovat?

Import z modulu:

```
import Data.List
import qualified Data.List as L
import Data.List (nub)
import Data.List hiding (nub)
```

Modul:

```
module Data.List where
```

(jméno souboru a cesta k modulu si musí odpovídat)

Balíčky a cesty

GHC vnímá koncept balíků, tyto je možné libovolně přidávat do balíků dostupných při kompilaci.

Nástroj `ghc-pkg`:

- `list`, `find-module module`, `describe pkg`
- `expose pkg`, `hide pkg`
- `register filename`, `unregister pkg`, `update filename`
  za `filename` se dosadí cesta ke konfiguračnímu souboru sestaveného balíku.

Je možné mít několik databází balíků (např. systémovou a lokální uživatelskou).

`cabal`

`cabal` je systém na jednoduché vytváření, sestavování, distribuci a instalaci balíčků a jejich závislostí.

Idea:

- Z každého samostatnějšího kusu software uděláme balíček!
  - dostane jméno
  - popíšeme závislosti
- Použijeme `cabal` na výrobu balíku!
  - sestavení
  - rozumná instalace a registrace pro `ghc`
- Balíček pěkně zabalíme a pošleme ostatním!
  - Hackage the haskell package database
- Stáhneme, sestavíme a nainstalujeme si i balíčky ostatních!
  - cabalem z Hackage

balik.cabal

```
name:                balik
version:             0.1.0.0
synopsis:            Venkovsky Balik
license:             AGPL-3
license-file:        LICENSE
author:              Jozef
maintainer:          jozef@vesnice.madarsko
category:            Web
build-type:          Simple
extra-source-files:  ChangeLog.md
cabal-version:       >=1.10

executable balik
  main-is:             Main.hs
```

balik.cabal

```
executable balik
  main-is:             Main.hs
  build-depends:
    base >=4.9,
    aeson,
    scotty,
    SHA,
    network,
    utf8-string

  hs-source-dirs:      src
```

Stáhneme databázi dostupných balíků:

```
cabal update
```

Něco nainstalujeme a zaregistrujeme to do aktuálně dostupného `ghc`:

```
cabal install acme-schoenfinkel
```

Stáhneme databázi dostupných balíků:

```
cabal update
```

Něco nainstalujeme a zaregistrujeme to do aktuálně dostupného `ghc`:

```
cabal install acme-schoenfinkel
cabal info acme-schoenfinkel
cabal list
```

Oddělené prostředí se občas hodí:

```
cabal sandbox init
cabal exec
cabal repl
```

Co když chceme `cabal` použít k výrobě nového balíku?

```
cd balik
cabal init
```

...vyplněním interaktivního formuláře vznikne `balik.cabal`

Instalace závislostí:

```
cabal install --only-dependencies
```

Sestavení:

```
cabal build
```

Spuštění:

```
cabal run
```

Instalace do aktuálního prostředí:

```
cabal install
```

Vyrobení HTML/TeX dokumentace:

```
cabal haddock
```

Vysázení pěkně barevného kódu pro publikaci:

```
cabal hscolour
```

Historie: Řešení některých odporných závislostních stromů:

`cabal new-build`, `cabal new-install`, `cabal new-run`, ...

Publikování balíku:

`cabal check` ...kontrola na běžné potíže

`cabal sdist` ...vyrobení `.tar.gz` balíku na manuální distribuci

`cabal upload` ...upload na Hackage

Jak se nestydět za publikovaný balík?

```
cabal install hlint

cabal install hindent

cabal check
```

`stack`

`stack` je zamražovač závislostí pro Haskell.

`stack`

`stack` je zamražovač závislostí pro Haskell.

stack

Hlavní výhody stacku:

- vlastní hackage (Stackage)
  - balíky a závislosti jsou pravidelně testované na kompatibilitu
  - většinou to částečně pomáhá
- v případě děsivých setupů se dá ušetřit dost času

Pravidla:

- Cabal balík: 0–1 knihoven, 0–N programů
  - popsáno v `balik.cabal`
- Stack projekt: 0–N cabalích balíků
  - projekt a celé zmražené prostředí popsáno v `stack.yaml`
  - metainformace balíku (tj. obsah generovaných cabalových souborů) v `package.yaml`
  - soubory `.cabal` stack generuje automaticky a nejde je měnit

stack

Cizí balík:

- `cd balik` ...adresář obsahuje `stack.yaml`
- `stack setup`
- `stack build` ...většinou stáhne správné GHC apod.
- `stack exec nazevprogramu`

Vlastní balík:

- `stack new nazevbaliku new-template`
- `stack test` ...spustí testy
- `stack repl` ...spustí `ghci` ve zamraženém prostředí

stack

Obecná rada: Pokud `stack` chcete použít pro svůj další projekt, nejspíš děláte něco špatně.

Proč `stack` nepoužívat:

- dostatečný, standardnější a spolehlivější nástroj na sestavování balíčků je `cabal`
- `stack` plýtvá zdroji
- `stack` třepí ekosystém
- výsledek je sice ‘přenositelný’, ale špatně udržovatelný
- způsob instalace `stack`u je podivný


DO NOT STACK
S-853, ULINE, 800-295-5510

Kdy `stack` asi budete muset použít?

- program má 9701734347576 závislostí a při změně libovolné se rozbije
- program je kritická součást nějaké infrastruktury, jejíž admin nechápe balíčky
- programátor neumí nebo nechápe UNIX
- programátor nemá čas nebo neumí udělat validní balíček
- velitel rozhodl
- cílový operační systém je podměrečný
- skočili jste časem do doby před vznikem `cabal new-build`

Tour de Prelude

Než začneme s čímkoliv větším, hodí se mít přehled o běžně používaných konstrukcích a knihovních funkcích:

- `Prelude`
- `System.IO`
- `Control.Monad`

Prelude

Prelude je ‘základ’ defaultně importovaný do všech modulů.

Obsahuje:

- standardní typy, typové třídy, související funkce a operátory
- základní věci pro práci s monádami
- základní IO

Výběr ze Prelude — IO

$$
\begin{aligned}
&\mathit{getLine}, \mathit{getContents} :: \mathit{IO\ String}\\
&\mathit{getChar} :: \mathit{IO\ Char}\\
&\mathit{readFile} :: \mathit{FilePath} \to \mathit{IO\ String}\\
&\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()\\
&\mathit{putStr}, \mathit{putStrLn} :: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{writeFile}, \mathit{appendFile} :: \mathit{FilePath} \to \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{readLn} :: \mathit{Read}\ a \Rightarrow \mathit{IO}\ a\\
&\mathit{print} :: \mathit{Show}\ a \Rightarrow a \to \mathit{IO}\ ()\\
&\mathit{interact} :: (\mathit{String} \to \mathit{String}) \to \mathit{IO}\ ()
\end{aligned}
$$

Výběr ze Prelude — seznamy

$head, last :: [a] \to a$
$tail, init :: [a] \to [a]$
$(+\!\!+) :: [a] \to [a] \to [a]$
$(!!) :: [a] \to Int \to a$
$elem :: [a] \to a \to Bool$
$length :: [a] \to Int$
$take, drop :: Int \to [a] \to [a]$
$takeWhile, dropWhile :: (a \to Bool) \to [a] \to [a]$
$lines, words :: String \to [String]$
$unlines, unwords :: [String] \to String$

Výběr ze Prelude — seznamy

$$map :: (a \to b) \to [a] \to [b]$$
$$filter :: (a \to Bool) \to [a] \to [a]$$
$$foldl :: (a \to x \to a) \to a \to [x] \to a$$
$$foldr :: (x \to a \to a) \to a \to [x] \to a$$
$$scanl :: (a \to x \to a) \to a \to [x] \to [a]$$
$$scanr :: (x \to a \to a) \to a \to [x] \to [a]$$
$$lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$$
$$reverse :: [a] \to [a]$$
$$replicate :: Int \to a \to [a]$$
$$repeat :: a \to [a]$$
$$cycle :: [a] \to [a]$$
$$iterate :: (a \to a) \to a \to [a]$$

Výběr ze Prelude — seznamy

```
import Data.List
intersperse :: a → [a] → [a]    -- vloží prvek mezi prvky seznamu
intercalate :: [a] → [[a]] → [a]    -- vloží seznam mezi seznamy
nub :: Eq a ⇒ [a] → [a]    -- "unique" bez Ord
sort :: Ord a ⇒ [a] → [a]    -- optimalizovaný mergesort
```

Výběr ze Prelude — seznamy

$$\mathit{nubBy} :: (a \to a \to \mathit{Bool}) \to [a] \to [a]$$
$$\mathit{sortOn} :: \mathit{Ord}\ b \Rightarrow (a \to b) \to [a] \to [a]$$
$$\mathit{sortBy} :: (a \to a \to \mathit{Ordering}) \to [a] \to [a]$$
$$\mathit{groupBy} :: (a \to a \to \mathit{Bool}) \to [a] \to [[a]]$$

Řazení od konce:

$$\mathit{sortOn}\ \mathit{reverse}\ [\texttt{"foo"}, \texttt{"bar"}, \texttt{"baz"}, \texttt{"oni"}] \quad \rightsquigarrow [\texttt{"oni"}, \texttt{"foo"}, \texttt{"bar"}, \texttt{"baz"}]$$

Hezké rozdělení slov ve větě:

$$\mathit{groupBy}\ ((\equiv)\ `\mathit{on}`\ \mathit{isLetter})\ \texttt{"Hello, Haskell!"}$$
$$[\texttt{"Hello"}, \texttt{", "}, \texttt{"Haskell"}, \texttt{"!"}]$$

$$\mathit{sortBy}\ (\mathit{comparing}\ \mathit{length} \diamond \mathit{compare})\ [\texttt{"bb"}, \texttt{"aaaa"}, \texttt{"bbbb"}, \texttt{"aa"}]$$
$$\rightsquigarrow [\texttt{"aa"}, \texttt{"bb"}, \texttt{"aaaa"}, \texttt{"bbbb"}]$$

Výběr ze Prelude — funkce

```
import Data.Function
const :: a → b → a    -- užitečný stavební blok
fix :: (a → a) → a    -- nekonečná sebeaplikace
(&) :: a → (a → b) → b    -- "shell pipe"
on :: (b → b → c) → (a → b) → a → a → c

import Data.Functor
(⟨&⟩) Functor f :: f a → (a → b) → f b    -- "actual shell pipe"
```

Typické použití *on*:

```
((+) `on` f) x y = f x + f y
sortBySecond = sortBy (compare `on` snd)
```

…porovnejte s *liftA2*

Výběr z Prelude — chyby

$\bot :: a$
error $:: [Char] \to a$
fail $::$ *MonadFail* $m \Rightarrow$ *String* $\to m\ a$
ioError $::$ *IOError* $\to$ *IO* a
userError $::$ *String* $\to$ *IOError*

fail je pozůstatek z původní definice monád, v některých případech se chová rozumně (např. *fail* v *Maybe* je *Nothing*). *ioError* je pěkný *fail* pro *IO*.

error je jako $\bot$ s chybovou hláškou navíc.

Výjimky

Chytání systémových výjimek:

```
import System.IO.Error

catchIOError :: IO a → (IOError → IO a) → IO a
tryIOError :: IO a → IO (Either IOError a)
```

Generičtější:

```
import Control.Exception

throw :: Exception e ⇒ e → a
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
try :: Exception e ⇒ IO a → IO (Either e a)
```

Generické výjimky

```
import Control.Monad.Catch

class Monad m ⇒ MonadThrow m where
  throwM :: Exception e ⇒ e → m a

class MonadThrow m ⇒ MonadCatch m where
  catch :: Exception e ⇒ m a → (e → m a) → m a
```

Instance by měly splňovat:

- $throwM\ e \gg f = throwM\ e$
- $catch\ (throwM\ e)\ f = f\ e$

Bracket pattern

Bracket zachycuje sémantiku vyrobení a odstranění zdroje:

$$bracket :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$$
$$bracket\_ :: IO\ a \to IO\ b \to IO\ c \to IO\ c$$

Typicky: *bracket openTheFile closeTheFile processTheFile*

Podobně:

$$bracketOnError :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$$
$$finally :: IO\ a \to IO\ b \to IO\ a$$
$$onException :: IO\ a \to IO\ b \to IO\ a$$

Chyby vs. výjimky

N.B. Existuje sémantický rozdíl mezi chybou (selháním programu) a výjimkou (situací, kterou programátor očekával, ale rozhodl se ji rozumně ošetřit podobně jako selhání).

Pravděpodobnost	Druh problému	Implementace
Nerealistická	Chyba	výskyt odvoditelný z dokumentace
Zanedbatelná	Chyba	$\bot$, *error* + dokumentace
Nízká, externí původ	Výjimka	*error*, *IOError*
Nízká, interní původ	Výjimka	*Exception*, *MonadFail*, *MonadCatch*
Očekávatelná	Data s výjimkou	*MonadFail*, *MonadCatch*, *Alternative*
Běžná	Standardní průběh programu	*Alternative*, *Either*, *Maybe*, *bool*

Soubory

V Haskellu se soubory drží za *Handle*.

```
import System.IO

openFile :: FilePath → IOMode → IO Handle
hClose :: Handle → IO ()
withFile :: FilePath → IOMode → (Handle → IO r) → IO r
```

Ostatní IO funkce mají varianty s *Handle*:

hFileSize, *hIsEOF*, *hFlush*, *hSeek*, *hTell*, *hReady*, *hGetChar*, *hGetContents*, *hGetLine*, *hPutChar*, *hPutStr*, *hPutStrLn*, *hPrint*, ...

import *Control.Monad*

$mapM :: (Monad\ m, Traversable\ t) \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)$
$mapM\_ :: (Monad\ m, Foldable\ t) \Rightarrow (a \to m\ b) \to t\ a \to m\ ()$

$sequence :: (Monad\ m, Traversable\ t) \Rightarrow t\ (m\ a) \to m\ (t\ a)$
$forever :: Applicative\ f \Rightarrow f\ a \to f\ b$

Bonusy:

- podobně: *zipWithM*, *replicateM*, *foldM*, *sequence_*
- *forM* a *forM_* je otočený *mapM*
- *when* a *unless* místo **if**
- Na dost věcí stačí aplikativní interface — pokud je to možné, používejte *traverse* a *traverse_*.
- Narozdíl od jiných jazyků, *IO* je normální monáda a kombinátory na ní fungují taky.

Kontejnery

Přehled

- Foldable & Traversable
- Maybe, Either, List
- Tree
- Sequence
- Array & Vector
- Graph
- Set, Map
- IntSet, IntMap

Data.Foldable

$$
\begin{aligned}
&\textbf{class } \mathit{Foldable}\ (t :: * \to *) \textbf{ where} \\
&\quad \mathit{Data.Foldable.fold} :: \mathit{Monoid}\ m \Rightarrow t\ m \to m \\
&\quad \mathit{foldMap} :: \mathit{Monoid}\ m \Rightarrow (a \to m) \to t\ a \to m \\
&\quad \mathit{foldr}, \mathit{foldr'} :: (a \to b \to b) \to b \to t\ a \to b \\
&\quad \mathit{foldl}, \mathit{foldl'} :: (b \to a \to b) \to b \to t\ a \to b \\
&\quad \mathit{foldr1}, \mathit{foldl1} :: (a \to a \to a) \to t\ a \to a \\
&\quad \mathit{Data.Foldable.toList} :: t\ a \to [a] \\
&\quad \mathit{null} :: t\ a \to \mathit{Bool} \\
&\quad \mathit{length} :: t\ a \to \mathit{Int} \\
&\quad \mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to t\ a \to \mathit{Bool} \\
&\quad \mathit{maximum}, \mathit{minimum} :: \mathit{Ord}\ a \Rightarrow t\ a \to a \\
&\quad \mathit{sum}, \mathit{product} :: \mathit{Num}\ a \Rightarrow t\ a \to a
\end{aligned}
$$

Utils: *and*, *or*, *all*, *any*, *foldlM*, *foldrM*

Data.Foldable

Používejte buď *foldl′* nebo *foldr*!

***foldl* občas (často) spotřebuje překvapivě hodně paměti.**


Vorsicht
Funktor
2 m Abstand halten

Data.Traversable

$$
\begin{aligned}
&\textbf{class } (\mathit{Functor}\ t, \mathit{Foldable}\ t) \Rightarrow \mathit{Traversable}\ t \textbf{ where} \\
&\quad \mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b) \\
&\quad \mathit{sequenceA} :: \mathit{Applicative}\ f \Rightarrow t\ (f\ a) \to f\ (t\ a) \\
&\quad \mathit{mapM} :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b) \\
&\quad \mathit{sequence} :: \mathit{Monad}\ m \Rightarrow t\ (m\ a) \to m\ (t\ a)
\end{aligned}
$$

Podobné: *traverse_*, *sequence_*, *mapM_*, *forM*, *mapAccumL*, *mapAccumR*

Data.Maybe

$$
\begin{aligned}
&\mathit{maybe} && :: b \to (a \to b) \to \mathit{Maybe}\ a \to b \\
&\mathit{fromMaybe} && :: a \to \mathit{Maybe}\ a \to a \\
&\mathit{fromJust} && :: \mathit{Maybe}\ a \to a \\
&\mathit{isJust} && :: \mathit{Maybe}\ a \to \mathit{Bool} \\
&\mathit{isNothing} && :: \mathit{Maybe}\ a \to \mathit{Bool} \\
&\mathit{listToMaybe} && :: [a] \to \mathit{Maybe}\ a \\
&\mathit{maybeToList} && :: \mathit{Maybe}\ a \to [a] \\
&\mathit{mapMaybe} && :: (a \to \mathit{Maybe}\ b) \to [a] \to [b] \\
&\mathit{catMaybes} && :: [\mathit{Maybe}\ a] \to [a]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{either} && :: (a \to c) \to (b \to c) \to \mathit{Either}\ a\ b \to c \\
&\mathit{fromLeft} && :: a \to \mathit{Either}\ a\ b \to a \\
&\mathit{fromRight} && :: b \to \mathit{Either}\ a\ b \to b \\
&\mathit{isLeft} && :: \mathit{Either}\ a\ b \to \mathit{Bool} \\
&\mathit{isRight} && :: \mathit{Either}\ a\ b \to \mathit{Bool} \\
&\mathit{lefts} && :: [\mathit{Either}\ a\ b] \to [a] \\
&\mathit{rights} && :: [\mathit{Either}\ a\ b] \to [b] \\
&\mathit{partitionEithers} && :: [\mathit{Either}\ a\ b] \to ([a], [b])
\end{aligned}
$$

Data.List

$$
\begin{aligned}
&\mathit{cycle} && :: [a] \to [a] \\
&\mathit{repeat} && :: a \to [a] \\
&\mathit{replicate} && :: \mathit{Int} \to a \to [a] \\
&\mathit{intersperse} && :: a \to [a] \to [a] \\
&\mathit{intercalate} && :: [a] \to [[a]] \to [a] \\
&\mathit{group} && :: \mathit{Eq}\ a \Rightarrow [a] \to [[a]] \\
&\mathit{nub} && :: \mathit{Eq}\ a \Rightarrow [a] \to [a] \\
&\mathit{inits} && :: [a] \to [[a]] \\
&\mathit{tails} && :: [a] \to [[a]] \\
&\mathit{sort} && :: \mathit{Ord}\ a \Rightarrow [a] \to [a] \\
&\mathit{sortBy} && :: (a \to a \to \mathit{Ordering}) \to [a] \to [a] \\
&\mathit{sortOn} && :: \mathit{Ord}\ b \Rightarrow (a \to b) \to [a] \to [a]
\end{aligned}
$$

Data.Tree

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ \{\mathit{rootLabel} :: a, \mathit{subForest} :: \mathit{Forest}\ a\}$$
$$\textbf{type}\ \mathit{Forest}\ a = [\mathit{Tree}\ a]$$

$$\begin{aligned}
\mathit{drawTree} &:: \mathit{Tree\ String} \to \mathit{String} \\
\mathit{drawForest} &:: \mathit{Forest\ String} \to \mathit{String}
\end{aligned}$$

$$\begin{aligned}
\mathit{levels} &:: \mathit{Tree}\ a \to [[a]] \\
\mathit{flatten} &:: \mathit{Tree}\ a \to [a]
\end{aligned}$$

$$\begin{aligned}
\mathit{foldTree} &:: (a \to [b] \to b) \to \mathit{Tree}\ a \to b \\
\mathit{unfoldTree} &:: (b \to (a, [b])) \to b \to \mathit{Tree}\ a
\end{aligned}$$

Data.Sequence

Sekvence je lepší seznam:

- indexování v $\mathcal{O}(\log n)$
- přístup k oběma koncům
- většina ne-globálních operací amortizovaně mezi $\mathcal{O}(1)$ až $\mathcal{O}(\log n)$

Operace navíc:

```
(⟨|)  :: a → Seq a → Seq a    -- přidávání prvku na konec
(|⟩)  :: Seq a → a → Seq a

data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a → ViewL a
viewr :: Seq a → ViewR a

(⟩⟨)  :: Seq a → Seq a → Seq a    -- O(log n) concat
```

Data.Array

Arraye mají indexy:

$$\begin{aligned}&\textbf{class } \mathit{Ord}\ a \Rightarrow \mathit{Ix}\ a \textbf{ where}\\ &\quad \mathit{range} \quad :: (a, a) \to [a]\\ &\quad \mathit{index} \quad :: (a, a) \to a \to \mathit{Int}\\ &\quad \mathit{inRange} \quad :: (a, a) \to a \to \mathit{Bool}\\ &\quad \mathit{rangeSize} :: (a, a) \to \mathit{Int}\end{aligned}$$

Typ arraye: *Array idx a*

Přístup do arraye podle indexu je $\mathcal{O}(1)$. Implementace je vestavěná v GHC, modelovaná jako funkce s omezenou kontinuální celočíselnou doménou.

$$
\begin{aligned}
&\mathit{array} && :: \mathit{Ix}\ i \Rightarrow (i, i) \to [(i, e)] \to \mathit{Array}\ i\ e \\
&\mathit{listArray} && :: \mathit{Ix}\ i \Rightarrow (i, i) \to [e] \to \mathit{Array}\ i\ e \\
&\mathit{assocs} && :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to [(i, e)] \\
&\mathit{bounds} && :: \mathit{Array}\ i\ e \to (i, i) \\
&\mathit{elems} && :: \mathit{Array}\ i\ e \to [e] \\
&(!) && :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to i \to e \\
&(//) && :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to [(i, e)] \to \mathit{Array}\ i\ e \\
&\mathit{ixmap} && :: (\mathit{Ix}\ j, \mathit{Ix}\ i) \Rightarrow \\
& && \qquad (i, i) \to (i \to j) \to \mathit{Array}\ j\ e \to \mathit{Array}\ i\ e
\end{aligned}
$$

Data.Vector

Vektory jsou arraye obalené trochu lepším API. Vyhledání prvku podle indexu je $\mathcal{O}(1)$, index je Int, a indexy začínají od nuly.

Operace podobné jako u arrayí:

```
length      :: Vector a → Int
(!)         :: Vector a → Int → a
(!?)        :: Vector a → Int → Maybe a

unsafeIndex :: Vector a → Int → a    -- rychlejší varianty s UB!
unsafeHead  :: Vector a → a
unsafeLast  :: Vector a → a

imap        :: (Int → a → b) → Vector a → Vector b

(//)        :: Vector a → [(Int, a)] → Vector a
update      :: Vector a → Vector (Int, a) → Vector a
update_     :: Vector a → Vector Int → Vector a → Vector a
```

Data.Vector.Mutable

Občas se hodí do pole smět náhodně chaoticky zapisovat. U immutable vektorů je ale výměna jednoho prvku $\mathcal{O}(n)$.

MVector s a si uchovává implicitní stav v monádě *s* (např. *IO*, *ST*), zápis do vektoru je v $\mathcal{O}(1)$.

```
new :: PrimMonad m ⇒ Int → m (MVector (PrimState m) a)
  {-unsafeNew neinicializuje paměť! :) -}
clone :: PrimMonad m ⇒
   MVector (PrimState m) a → m (MVector (PrimState m) a)
read :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → m a
write :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → a → m ()
modify :: PrimMonad m ⇒
   MVector (PrimState m) a → (a → a) → Int → m ()
swap :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → Int → m ()
```

```
import Control.Monad
import qualified Data.Vector.Mutable as V

bublsort mv = sequence_
   [do [left, right] <- traverse (V.read mv) [i, i + 1]
       when (left > right) $ V.swap mv i (i + 1)
    | let n = V.length mv, _ <- [0 .. n - 1], i <- [0 .. n - 2]]

demoVector = do v <- V.new 5 :: V.IOVector Int
                zipWithM (V.write v) [0 ..] [5, 3, 2, 5, 1]
                return v

printV v = forM_ [0 .. V.length v - 1] $ (>>=print) . V.read v

main = do v <- demoVector
          bublsort v
          printV v
```

Data.Graph

```
type Vertex  = Int
type Edge    = (Vertex, Vertex)
type Bounds  = (Vertex, Vertex)
type Table a = Array Vertex a

buildG     :: Bounds → [Edge] → Graph
vertices   :: Graph → [Vertex]
edges      :: Graph → [Edge]

indegree   :: Graph → Table Int
outdegree  :: Graph → Table Int

path       :: Graph → Vertex → Vertex → Bool
reachable  :: Graph → Vertex → [Vertex]
components :: Graph → Forest Vertex
```

```
graphFromEdges :: Ord key ⇒ [(node, key, [key])] →
                             (Graph,
                              Vertex → (node, key, [key]),
                              key →    Maybe Vertex)
graphFromEdges'   -- nevrací mapu klíčů
transposeG :: Graph → Graph
topSort    :: Graph → [Vertex]   -- topologické uspořádání
dfs        :: Graph → [Vertex] → Forest Vertex   -- zakořeněné kostry
dff        :: Graph → Forest Vertex   -- tosamé pro všechny kořeny
scc        :: Graph → Forest Vertex   -- souvislé komponenty
bcc        :: Graph → Forest [Vertex]   -- 2-souvislé komponenty
```

Množiny a mapy

Standardní stromovité kontejnery: *Map*, *IntMap*, *Set*, *IntSet*

- Int-varianty jsou optimalizovanější
- Jsou dostupné i Lazy varianty (většinou je vhodnější použít Strict)
- Běžná metoda importu s kvalifikátorem zamezuje kolizi jmen:

```
import qualified Data.Map.Strict as M
```

Data.Set, Data.Map

$$
\begin{array}{lll}
M.fromList :: Ord\ k \Rightarrow & [(k,a)] \to M.Map\ k\ a \\
S.fromList :: Ord\ a \Rightarrow & [a] \to S.Set\ a \\
M.assocs :: & M.Map\ k\ a \to [(k,a)] \\
M.elems :: & M.Map\ k\ a \to [a] \\
S.elems :: & S.Set\ a \to [a]
\end{array}
$$

$$
S.insert, S.delete :: Ord\ a \Rightarrow a \to S.Set\ a \to S.Set\ a
$$

$$
\begin{array}{lll}
M.insert :: Ord\ k \Rightarrow & k \to a & \to M.Map\ k\ a \to M.Map\ k\ a \\
M.delete :: Ord\ k \Rightarrow & k & \to M.Map\ k\ a \to M.Map\ k\ a \\
M.adjust :: Ord\ k \Rightarrow & (a \to a) \to k & \to M.Map\ k\ a \to M.Map\ k\ a \\
(M.!) :: Ord\ k \Rightarrow & M.Map\ k\ a \to k & \to a \\
(M.!?) :: Ord\ k \Rightarrow & M.Map\ k\ a \to k & \to Maybe\ a
\end{array}
$$

$$
S.union, S.difference, S.intersection :: Ord\ a \Rightarrow S.Set\ a \to S.Set\ a \to S.Set\ a
$$

Množiny a mapy (bonusy)

V balíku `unordered-containers` je navíc *HashSet* a *HashMap*

NB.: Hashování v kontejnerech je téměř vždy granát do vlastních bot; v případě funkcionálního programování to platí exponenciálně víc. Pokud jste zvládli DS 2, PDS a Vývoj vysoce výkonného SW a ještě pořád si myslíte, že to je dobrý nápad, můžete je použít.

Implementace:

class *Hashable a* **where**
  hashWithSalt :: *Int* $\to$ *a* $\to$ *Int*
member :: (*Eq a*, *Hashable a*) $\Rightarrow$ *a* $\to$ *HashSet a* $\to$ *Bool*

...

Set není funktor!

V jazycích bez mutovatelných hodnot tradiční datové struktury nefungují úplně dobře!

Typický příklad: zápis jednoho prvku do vektoru musí zkopírovat celý blok dat (co kdyby původní vektor ještě nekdo potřeboval).

Některé struktury fungují pro některá použití částečně dobře:

- Zápis do hlavy/kořene v seznamu/stromě: $\mathcal{O}(1)$
- Obecně: zápis v hloubce h přepíše $\mathcal{O}(h)$ dat.

Funkcionální datové struktury

Funkcionální datové struktury minimalizují h.

- velké bloky dat snižují h ale zvyšují konstantní overhead
- h je zdola omezené velikostí bloků a počtem uložených prvků

$$n_h = \mathcal{O}(B^h)$$

Běžné optimalizované struktury využívají např.:

- specializované části struktur
- optimistické předpoklady o používání struktury

Funkcionální fronta

Chceme potenciálně nekonečnou FIFO frontu s $\mathcal{O}(1)$ enqueue a dequeue.

Problémy:

- obousměrný seznam v immutable datech nejde vyrobit.
- změna jednosměrného seznamu na konci stojí $\mathcal{O}(n)$ *alokací*

Funkcionální fronta

Řešení: oddělíme polovinu fronty na čtení a polovinu fronty na zápis.

$$\textbf{data}\ FQueue\ a = FQueue\ \{fqReadEnd :: [a], fqWriteEnd :: [a]\}$$

$$fqEnqueue\ a\ q = q\ \{fqWriteEnd = a : fqWriteEnd\ q\}$$

$$fqRefill\ (FQueue\ [\,]\ x) = FQueue\ (reverse\ x)\ [\,]$$
$$fqRefill\ q = q$$

$$fqDequeue\ q = \textbf{case}\ fqRefill\ q\ \textbf{of}$$
$$\quad r@(FQueue\ [\,]\ \_) = Nothing$$
$$\quad FQueue\ (r : rs)\ w = Just\ (r, FQueue\ rs\ w)$$

Funkcionální fronta

Enqueue je viditelně $\mathcal{O}(1)$.

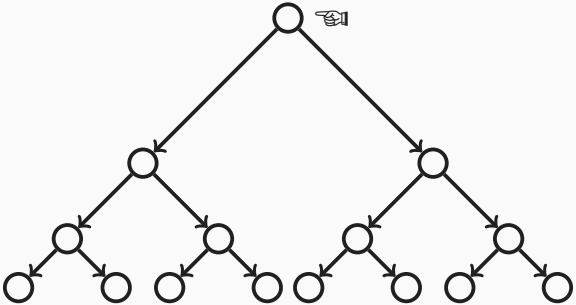
Dequeue je amortizovaně $\mathcal{O}(1)$. Pro každý průchozí prvek proběhnou operace:

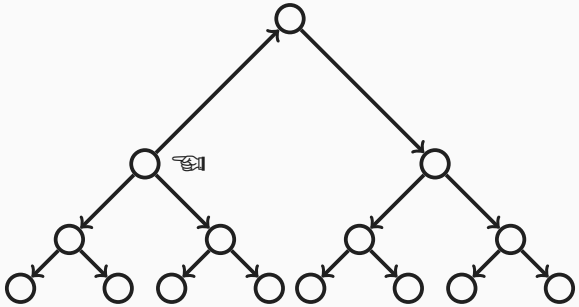
- připojení na začátek seznamu
- reverse (konstantní cena za prvek)
- odpojení ze začátku seznamu

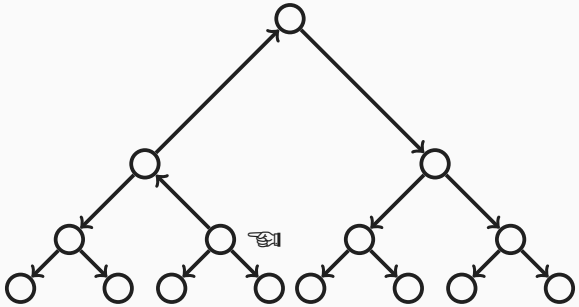
Zipper Tree

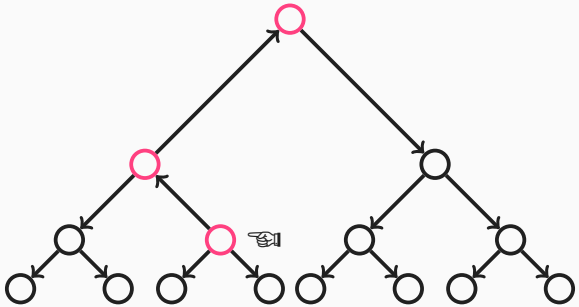
Zipper je generická konstrukce jak z normální datové struktury udělat vhodnou pro funkcionální programování. Princip si ukážeme na stromech.

- Normální stromy jsou reprezentované odkazem na kořen, hloubka všech operací se měří seshora.
- Zipper tree jsou reprezentované odkazem na libovolný prvek, ze kterého jdou ukazatele jak nahoru (až do kořene) tak dolů (až do listů).









Páteř je nutné specializovat.

$$
\begin{aligned}
&\textbf{data}\ \mathit{BTree}\ a = \mathit{BNode}\ (\mathit{BTree}\ a)\ a\ (\mathit{BTree}\ a) \mid \mathit{Nil} \\
&\textbf{data}\ \mathit{ZSpine}\ a = \mathit{RootR}\ a\ (\mathit{BTree}\ a) \\
&\qquad\qquad\qquad\quad \mid \mathit{RootL}\ (\mathit{BTree}\ a)\ a \\
&\qquad\qquad\qquad\quad \mid \mathit{SpineR}\ (\mathit{ZSpine}\ a)\ a\ (\mathit{BTree}\ a) \\
&\qquad\qquad\qquad\quad \mid \mathit{SpineL}\ (\mathit{BTree}\ a)\ a\ (\mathit{ZSpine}\ a) \\
&\textbf{data}\ \mathit{ZBTree}\ a = \mathit{Root}\ (\mathit{BTree}\ a) \mid \mathit{NonRoot}\ (\mathit{BTree}\ a)\ (\mathit{ZSpine}\ a) \\
&\mathit{zUp}, \mathit{zLeft}, \mathit{zRight} :: \mathit{ZBTree}\ a \to \mathit{ZBTree}\ a
\end{aligned}
$$

Implementace jde odvodit z původní struktury mechanicky pomocí *derivace na termech*. Koncept je aplikovatelný na jakoukoliv strukturu.

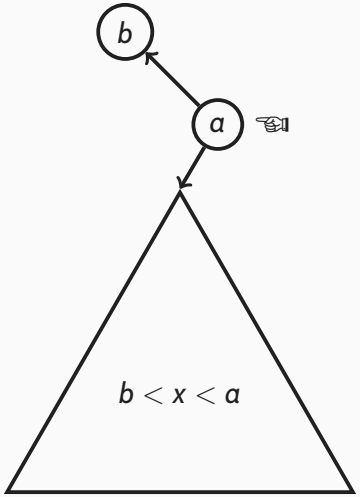
✎Gerard Huet, “Functional Pearl: The Zipper.” *Journal of Functional Programming* 7 (5), pp. 549–554 (1997).

Standardní implementace operace *find* by vystoupala ke kořeni a použila hledání z BVS.

Úsporná implementace se při stoupání může zastavit dříve:

- hledaný prvek má klíč x
- aktuální uzel je
  - $u = \mathit{NonRoot}\ (\mathit{BNode}\ \mathit{lst}\ a\ \_)\ (\mathit{SpineL}\ \_\ b\ \_)$ nebo
  - $u = \mathit{NonRoot}\ (\mathit{BNode}\ \mathit{lst}\ a\ \_)\ (\mathit{RootL}\ \_\ b)$
- platí $b < x < a$
- uzel s x existuje $\iff$ uzel je v podstromu *lst*

(platí i symetricky obráceně, s mírnou úpravou i pro intervaly)


b
a
☜
b < x < a

Je to rychlé?

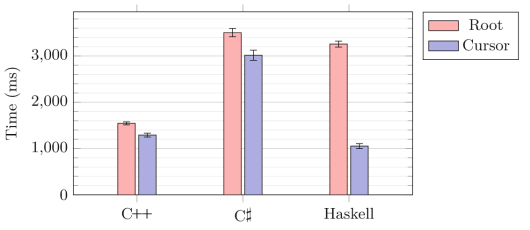


Fig. 6. 2-3 Tree insertion performance

✎Šefl, Vít. "Performance Analysis of Zippers." In *Declarative Programming and Knowledge Management*, pp. 215-229. Springer, Cham, 2019. `https://arxiv.org/pdf/1908.10926.pdf`

Bonus: Alokace a GC-shrabání paměti v *nursery* ve standardní implementaci GHC RTS je oproti `malloc`u výrazně rychlejší.

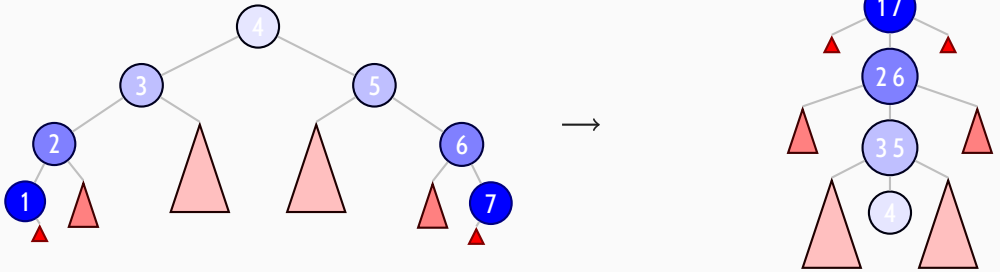
Pozor: To ani zdaleka neplatí univerzálně pro všechny GC.

Finger tree

✎Ralf Hinze and Ross Paterson, “Finger trees: a simple general-purpose data structure.” *Journal of Functional Programming* 16:2 (2006), pp. 197–217.

Konstrukce: Zipper z ‘obou stran’ — strom je rozdělený podle podstromů od kraje, umožňuje rychlý přístup ke krajním prvkům a rychlé spojování/rozpojování.

Finger tree (myšlenka)



2-3 Finger tree (Hinze&Paterson, 2006)

$$\begin{aligned}
&\textbf{data}\ \mathit{FingerTree}\ a = \mathit{Empty} \\
&\qquad\qquad\qquad\qquad \mid \mathit{Single}\ a \\
&\qquad\qquad\qquad\qquad \mid \mathit{Deep}\ (\mathit{Digit}\ a)\ (\mathit{FingerTree}\ (\mathit{Node}\ a))\ (\mathit{Digit}\ a) \\
&\textbf{data}\ \mathit{Digit}\ a = \mathit{One}\ a \mid \mathit{Two}\ a\ a \mid \mathit{Three}\ a\ a\ a \mid \mathit{Four}\ a\ a\ a\ a \\
&\textbf{data}\ \mathit{Node}\ a = \mathit{Node2}\ a\ a \mid \mathit{Node3}\ a\ a\ a
\end{aligned}$$

Sekvence (*Data.Sequence*)

Požadavky: rychlý přístup k oběma koncům, log-*n* nalezení *k*-tého prvku.

Implementace: 2-3 finger tree označkovaný velikostí podstromů.

Tradeoff: Sekvence nemůžou být nekonečné (narozdíl od seznamů).

Kompilujeme Haskell

Průběh kompilace

1. Parsování
2. **Typecheck**
3. Desugaring (převod do jazyka Core)
4. Optimalizace (“simplifier”)
5. Převod do STG, optimalizace
6. Převod do Cmm nebo LLVM, optimalizace
7. sestavení
8. linkování s RTS

Simon Peyton Jones & Simon Marlow

Jak typecheckovat funkcionální kód?

Rozdíl oproti C-like: postupování ‘po směru výpočtu’ nám toho moc neřekne.

- literály můžou mít libovolné typy, defaultovat a “kdyžtak později kovertovat” nelze
- občas je potřeba odhadnout typ parametru funkce (`auto`-parametry v C++)
- v rekurzivních případech dopředné odvozování selže

Popíšeme si Hindley-Milnerův systém a jeho rozšíření o typové třídy.

(c) Alonzo Church 1930

$$V ::= a|b|c|\cdots|x_\infty$$
$$\Lambda ::= V|\Lambda\Lambda|(\lambda V.\Lambda)$$

Kupodivu:

- Systém je T-úplný
- Díky jednoduchosti je to skvělý model pro “lidské” programovací jazyky.

Lambda kalkulus neumí hodnoty?

```
#haskell
20:30 < Ariakenom> lambda calculus has both
                   functions and values,
                   because functions
                   are values
```

Alonzo Church

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add\,1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add\,1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add}\,\mathbf{1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

$\mathbf{decr} = (\lambda n.\mathbf{right}(n\ \mathbf{incr2}\ (\mathbf{pair}\ \mathbf{0}\ \mathbf{0}))),$
$\mathbf{incr2} = (\lambda p.\mathbf{pair}\ (\mathbf{incr}\ (\mathbf{left}\ p))\ (\mathbf{if}\ (\mathbf{Z?}\ (\mathbf{left}\ p))\ \mathbf{0}\ (\mathbf{incr}\ (\mathbf{right}\ p))))$

λ-kuchařka

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add\ 1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

$\mathbf{decr} = (\lambda n.\mathbf{right}(n\ \mathbf{incr2}\ (\mathbf{pair\ 0\ 0}))),$
$\mathbf{incr2} = (\lambda p.\mathbf{pair}\ (\mathbf{incr}\ (\mathbf{left}\ p))\ (\mathbf{if}\ (\mathbf{Z?}\ (\mathbf{left}\ p))\ \mathbf{0}\ (\mathbf{incr}\ (\mathbf{right}\ p))))$

Rekurze bez možnosti použít vlastní definici:

$\mathbf{facF} = (\lambda fn.\mathbf{if}(\mathbf{Z?}\ n)\ \mathbf{1}\ (\mathbf{mul}\ n\ (f(\mathbf{decr}\ n))))$

$\mathbf{fac} = Y\ \mathbf{facF}, Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))), \infty = (\lambda fx.Yfx) = Y$

Věta o pevném bodě pro λ-kalkulus

$$(\forall F)(\exists X)\, X = F(X)$$

Dk.:

Věta o pevném bodě pro λ-kalkulus

$$(\forall F)(\exists X)\, X = F(X)$$

Dk.:

- **Pomůcka:** $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$

Věta o pevném bodě pro λ-kalkulus

$$(\forall F)(\exists X)\, X = F(X)$$

Dk.:

- **Pomůcka:** $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$
- $X = YF, \quad YF = F(YF).$ ✌

Věta o pevném bodě pro λ-kalkulus

$$(\forall F)(\exists X)\, X = F(X)$$

Dk.:

- **Pomůcka:** $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$
- $X = YF, \quad YF = F(YF).$ ✌

Pozorování: $YF = F(YF) = F(F(YF)) = F^{\infty}(YF)$.

Fixed point combinator

Y se tradičně jmenuje ‘fixed point combinator’ kvůli podobnosti s analytickými větami o pevném bodě:

- Analýza: najdu x tak, že po spočítání (zachovávajícím ekvivalenci) z $f(x)$ vyjde x, tudíž $x = f(x)$.
- λ: najdu X tak, že po spočítání (zachovávajícím ekvivalenci) vyjde $F(X)$, tudíž $X = F(X)$.

Teorie vyčíslitelnosti fixpointu říká ‘while cyklus’

- $F^{\infty}(X)$ může cyklit donekonečna
- podmínka v F může cyklus kdykoliv zastavit

D. Ú.: Na faktoriál přece není potřeba ani while-cyklus, ani dekrementace! Vyrobte **fac** bez **decr** a Y!

Fixpoint v Haskellu

let *fix f* $=$ *f* (*fix f*)

Cvičení (k vysvětlení výsledku):

- *fix* $(\lambda f\ i \to i : f\ (i + 1))\ 1$
- *fix* $(1:)$
- *fix show*
- *fix* (*scanl* $(+)\ 0 \cdot (1:)$)

Jak se nezacyklit

Pokud jde λ-term vůbec dostat do normální formy, určitě se do ní jde dostat pomocí vyhodnocování nejlevějšího redexu.

- $(\lambda x.xx)\ (\lambda x.xx)$
- $K'\ ((\lambda x.xx)\ (\lambda x.xx))(\lambda x.x)$

Tomu se říká lazy vyhodnocování.

Bonus: Kombinátorový kalkulus

Uzavřeným termům se říká kombinátory. (F, K, S, I, ...)

Věta (SKI kalkulus): Každý λ-term lze vyjádřit jako výraz složený pouze z aplikací kombinátorů

$$S = \lambda xyz.xz(yz)$$
$$K = \lambda xy.x$$

Např.:

$$I = SKK$$
$$K' = SK$$
$$F = S(K(SI))K \qquad \texttt{-flip}$$
$$Y = S(K(SII))(S(S(KS)K)(K(SII)))$$
$$\vdots$$

Bonus: Point-free

Trochu podobně se každý Haskellový term (bez speciální syntaxe) dá zbavit abstrakcí, výsledkem je tzv. point-free forma.

Konverze jednoduchých výrazů podle výskytu abstrahované proměnné:

$\lambda x \to x$	id
$\lambda x \to a$	$const\ a$
$\lambda x \to a\ x$	a
$\lambda x \to a\ (b\ x)$	$a \cdot b$
$\lambda x \to (a\ x)\ b$	$flip\ a\ b$
$\lambda x \to (a\ x)\ (b\ x)$	$a\ \langle * \rangle\ b$
$\lambda x \to \lambda y \to a$	rekurzivně

Složitější termy obsahující x 'někde hluboko' je možné η-expandovat na některý z jednodušších případů a vyřešit rekurzivně.

Jde nějak staticky zjistit, co daný λ-term dělá?

Nápad: zkusíme odvodit typ a uvidíme.

Jazyk typů:

$$V_T ::= \alpha | \beta | \gamma | \cdots | \tau_\infty$$
$$T ::= V_T | T \to T$$

Tradiční zápis: $\Gamma \vdash M : \tau$

Např.: $\vdash I : \tau \to \tau \quad \vdash K : \alpha \to \beta \to \alpha \quad \vdash \mathbf{n} : (\sigma \to \sigma) \to \sigma \to \sigma$
$\vdash \mathbf{if} : (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha \quad \{x : \alpha \to \beta, y : \alpha\} \vdash xy : \beta$

Mechanické odvození typů v STLC

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{\Gamma \vdash M : \pi \to \rho \quad \Gamma \vdash N : \pi}{\Gamma \vdash MN : \rho} \text{ (App)}$$

$$\frac{\Gamma \cup \{v : \pi\} \vdash M : \rho}{\Gamma \vdash (\lambda v.M) : (\pi \to \rho)} \text{ (Abs)}$$

Pravidla se občas nazývají postupně Axiom, $\to$-eliminace a $\to$-zavedení.

Slabá normalizace:

$$(\forall M \in \Lambda) \qquad M : \tau \implies M \text{ má N.F.}$$

Nevýhoda: většina užitečného programování typ v STLC nemá, ale “shodou okolností” má normální formu. (typicky: Y)

Slabá normalizace:

$$(\forall M \in \Lambda) \qquad M : \tau \implies M \text{ má N.F.}$$

Nevýhoda: většina užitečného programování typ v STLC nemá, ale “shodou okolností” má normální formu. (typicky: Y)

Existují typové systémy se **silnou normalizací** (tj. kde právě všechny zastavující programy mají typ), problém odvození typu v nich ale není rozhodnutelný. Např. $\lambda_{\cap}^{\rightarrow}$

Související: rozdíl mezi PRF, ORF a ČRF, viz. vyčíslitelnost.

Girard (1972) a nezávisle Reynolds (1974) z různých důvodů zavedli polymorfní typování.

$$T ::= V_T | T \to T | \forall V_T . T$$

Extra pravidla:

$$\frac{\Gamma \vdash M : \sigma \quad \alpha \notin \Gamma}{\Gamma \vdash M : (\forall \alpha . \sigma)} \text{ (Generalizace)}$$

$$\frac{\Gamma \vdash M : (\forall \alpha . \sigma)}{\Gamma \vdash M : \sigma[\alpha := \tau]} \text{ (Specializace)}$$

Jean-Yves Girard & John C. Reynolds

Výhody:

- $I : \forall\alpha.\alpha \to \alpha$, jde použít víckrát
- $(\lambda x.xx) : (\forall\beta.(\forall\alpha.\alpha) \to \beta)$

Nevýhody:

- $(\lambda x.xx) : (\forall\beta.(\forall\alpha.\alpha) \to (\beta \to \beta))$
- $(\lambda x.xx) : (\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$
- nerozhodnutelná inference

  …

Použití generalizace je příliš nedeterministické. Zkusme to rozumně.

Hindley-Milnerův systém

Zavedeme novou jazykovou konstrukci **let** $v = d$ **in** e, jejímž výsledkem bude polymorfní v použitelné v e.

Generalizaci v typech můžeme omezit na “vrchní úroveň”.

$$\Lambda ::= V | \Lambda\Lambda | (\lambda V.\Lambda) | \textbf{let } V = \Lambda \textbf{ in } \Lambda$$

$$\mathcal{M} ::= V_T | \mathcal{M} \to \mathcal{M} \qquad \text{(monotypy)}$$

$$\mathcal{P} ::= \mathcal{M} | \forall V_T.\mathcal{P} \qquad \text{(polymorfní typy)}$$

Odpovídající úprava STLC:

$$\frac{x : \tau \in \Gamma \quad \tau \in \mathcal{M}}{\Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{x : \tau \in \Gamma \quad \tau \in \mathcal{P}}{\Gamma \vdash x : \text{instantiate}(\tau)} \text{ (Var-Inst)}$$

$$\frac{\Gamma \vdash a : \pi \to \rho \quad \Gamma \vdash b : \pi}{\Gamma \vdash a\, b : \rho} \text{ (App)}$$

$$\frac{\Gamma \cup \{v : \pi\} \vdash e : \rho}{\Gamma \vdash (\lambda v.e) : (\pi \to \rho)} \text{ (Abs)}$$

$$\frac{\Gamma \vdash d : \tau \quad \Gamma \cup \{v : \text{quantify}(\tau)\} \vdash e : \sigma}{\Gamma \vdash (\textbf{let}\ v = d\ \textbf{in}\ e) : \sigma} \text{ (Let-Gen)}$$

Běžná implementace H-M

- Proměnné zachycené v **let** přiřadíme nejobecnější možný typ.
- Pokud někde použijeme polymorfní proměnnou, typ instancujeme na monomorfní typ doplněním čerstvých typových proměnných za polymorfní.

Př.: **let** *I* $= (\lambda x.x)$ **in** *I I*

- $(\lambda x.x) : \alpha \to \alpha$
- *I* : $\forall\alpha.\alpha \to \alpha$
- Druhé I: $\beta \to \beta$
- První I: $\gamma \to \gamma$, posléze $(\beta \to \beta) \to (\beta \to \beta)$
- Celý výraz: $\beta \to \beta$

Bez instancování není polymorfismu!

$$id\ id :: a \to a$$

ale:

$$(\lambda x \to x\ x)\ id$$

Occurs check: cannot construct the infinite type: $t \sim t \to t$

Typové třídy

Rozšíření H-M pro podporu typových tříd:

- kvalifikované polymorfní typy
  kvalifikace je efektivně množina tvrzení o typových proměnných (ve tvaru predikátů)
- kontroly typových tříd při specializaci
- složitější hledání nejobecnějšího typu
  - v rekurzivním případě to je nerozhodnutelný problém
  - některé odvozené predikáty patří kontextu
    (typicky ve vnořených definicích)
  - některé typy dovolují nejednoznačné chování
    $f = show \cdot read$

✎Mark P. Jones, "Typing Haskell In Haskell." `http://web.cecs.pdx.edu/~mpj/thih/`

Rekurzivní polymorfismus

Typ vzájemně polymorfních rekurzivních definic není možné automaticky odvodit!

$$\textbf{data}\ \mathit{Seq}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{Seq}\ (a, a))$$

$$\begin{aligned}&\mathit{seqmap}\ f\ \mathit{Nil} = \mathit{Nil}\\ &\mathit{seqmap}\ f\ (\mathit{Cons}\ x\ xs) =\\ &\quad \mathit{Cons}\ (f\ x)\ (\mathit{seqmap}\ (\lambda(a, b) \to (f\ a, f\ b))\ xs)\end{aligned}$$

Error: cannot construct the infinite type...

Oprava: typ připíšeme ručně.

$$\mathit{seqmap} :: (a \to b) \to \mathit{Seq}\ a \to \mathit{Seq}\ b$$

Defaulting

$$f = show \cdot read$$
$$f :: String \to String$$

...ale co se vlastně stane uvnitř?

Podobně:

$$main = print\ 5$$

Vypíše se Int, Integer, Float, nebo něco jiného?

Defaulting: Kompilátor v takových případech vyzkouší přednastavené defaultní typy.

Běžně např: **default** (*Integer*, *Double*)

Defaulting jde pro modul vypnout: **default** ()

Defaulting zevnitř

$$
\begin{aligned}
&f\ x = show\ (read\ x)\\
&read :: Read\ a \Rightarrow String \to a\\
&x :: String\\
&(read\ x) :: Read\ a \Rightarrow a\\
&show :: Show\ a \Rightarrow a \to String\\
&show\ (read\ x) :: (Read\ a, Show\ a) \Rightarrow String\\
&f :: (Read\ a, Show\ a) \Rightarrow String \to String
\end{aligned}
$$

Nepoužitá kvalifikovaná proměnná označuje nějakou nejednoznačnost (ambiguity).

Nejednoznačnosti se *odstraní* vyhledáním prvního typu ze seznamu **default**, který vyhovyje všem predikátům (případně chybou).

Typový systém Haskellu

Haskell navíc obsahuje poměrně velké množství rozšíření:

- víceparametrové typové třídy
- DataKinds, “dependent” types
- explicitní typovou rovnost
- type families
- lineární typy
- …

Core

Kód který prošel typecheckem se transformuje na Core.

```
data Expr b = Var b
            | Lit    Literal
            | App    (Expr b) (Arg b)
            | Lam    b (Expr b)
            | Let    (Bind b) (Expr b)
            | Case   (Expr b) b Type [Alt b]
            | Cast   (Expr b) Coercion
            | Tick   (Tickish Id) (Expr b)
            | Type   Type
            | Coercion Coercion
```

(`CoreSyn.hs:255-266`)

Core

Výhody:

- na jednoduchý kód se jednoduše píšou transformace
  - DCE
  - Inlining
  - Specializace
- jde typecheckovat (tj. ověřovat správnost transformací)

Zajímavosti:

- z kvantifikace typů se stávají parametry funkcí
- typové třídy se do Core ani nedostanou (jsou předělané na slovníkové funkce)

STG

Spineless Tagless Graph machine je VM pro vykonávání Haskellu.

VM obsahuje:

- registry
- stack
  - Haskell nemá viditelný stack!
  - používaný na continuations
- heap
  - data
  - thunks
  - functions
  - continuations

Hodnoty na heapu jsou propojené pointery.

STG a laziness

Data mají mnoho podob:

- unboxed — P. O. D.
- boxed, unlifted — mají ‘obal’ který místo nich může obsahovat thunk, ale nemůžou být $\perp$.
- boxed, lifted — mají ‘obal’ a navíc můžou být $\perp$ (vyrobit výjimku).

Život na STG:

- funkce je pointer na kód s aritou a uzávěrem
- přidáním dostatku argumentů se funkce změní na thunk
- forcováním se thunk (občas) změní na data

Laziness je důsledkem oddělení aplikace funkcí a vlastního výpočtu.

Trocha šetření

Aby se funkce zbytečně nepočítaly víckrát než je potřeba, Haskell pro každý let-binding v kódu vygeneruje pouze jeden thunk při každém vstupu do scope.

Když nějaká část programu thunk poprvé spočítá, nahradí ho výsledkem, ten můžou dál používat i ostatní části programu.

Trocha šetření

Aby se funkce zbytečně nepočítaly víckrát než je potřeba, Haskell pro každý let-binding v kódu vygeneruje pouze jeden thunk při každém vstupu do scope.

Když nějaká část programu thunk poprvé spočítá, nahradí ho výsledkem, ten můžou dál používat i ostatní části programu.

Trocha šetření

Aby se funkce zbytečně nepočítaly víckrát než je potřeba, Haskell pro každý let-binding v kódu vygeneruje pouze jeden thunk při každém vstupu do scope.

Když nějaká část programu thunk poprvé spočítá, nahradí ho výsledkem, ten můžou dál používat i ostatní části programu.

Trocha šetření

Aby se funkce zbytečně nepočítaly víckrát než je potřeba, Haskell pro každý let-binding v kódu vygeneruje pouze jeden thunk při každém vstupu do scope.

Když nějaká část programu thunk poprvé spočítá, nahradí ho výsledkem, ten můžou dál používat i ostatní části programu.

Výsledek:

- Let-binding bez parametrů se vždycky vypočítá jen jednou
- Funguje $\mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})$
- monomorphism restriction

Odstranění implicitního scope: Lifting

Asembler, STG ani Cmm nemají scope. Lifting je jednoduchá transformace, která scope doplní:

$$\begin{aligned} &\mathit{solutions}\ a\ b\ c = \mathit{map}\ (/(2 * a))\ [-b + sD, -b - sD] \\ &\quad \textbf{where}\ sD = \mathit{sqrt}\ (b * b - 4 * a * c) \end{aligned}$$

Doplnění argumentů:

$$\begin{aligned} &\mathit{solutions}\ a\ b\ c = \mathit{map}\ (/(2 * a))\ [-b + sD\ a\ b\ c, -b - sD\ a\ b\ c] \\ &\quad \textbf{where}\ sD\ a\ b\ c = \mathit{sqrt}\ (b * b - 4 * a * c) \end{aligned}$$

Lift:

$$\begin{aligned} &sD\ a\ b\ c = \mathit{sqrt}\ (b * b - 4 * a * c) \\ &\mathit{solutions}\ a\ b\ c = \mathit{map}\ (/(2 * a))\ [-b + sD\ a\ b\ c, -b - sD\ a\ b\ c] \end{aligned}$$

Odstranění přímé rekurze a stacku

Konverze na Continuation-Passing-Style:

$$
\begin{aligned}
&\mathit{sD}\ a\ b\ c\ \mathit{cont} = \mathit{cont}\ (\mathit{sqrt}\ (b * b - 4 * a * c)) \\
&\mathit{solutions}\ a\ b\ c\ \mathit{cont} = \\
&\quad \mathit{sD}\ a\ b\ c\ \$\ \lambda \mathit{d1} \rightarrow \\
&\quad \mathit{map}\ (/(2 * a))\ [-b + \mathit{d1}, -b - \mathit{d1}]\ \mathit{cont}
\end{aligned}
$$

(*sqrt* a ostatní číselné operace považujeme za primitivní)

Continuation je víceméně jen pointer na kód.

Low-level

- STG se (už poměrně jednoduše) převede na Cmm
  - Cmm je C s garbage collectorem, ale bez složitých typů a pořádných pointerů
- Cmm jde skompilovat podobně jako C na modul `.o`
- Výsledek se slinkuje s RTS, který doplní:
  - `main`
  - Cmm rutiny pro GC (alokaci apod.)
  - Cmm rutiny pro IO
  - implementaci spodních vrstev GHC.Prim

Low-level: moduly

Modularizace funguje obdobně jako u C/C++:

- `.hs` odpovídá zdrojovému kódu `.c` a `.cpp`
- `.hi` odpovídá vygenerovaným hlavičkovým souborům
- `.o` odpovídá Cčkovému `.o`, jde předělat na `.so` nebo `.a` (případně `.dll` a `.lib`) a standardně linkovat

Typický problém: Změny v `.hi` (i špatné) se mohou propagovat do jiných `.hi`. (Tradiční řešení: `cabal nuke`)

Bonus: použití STLC

$$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \_$$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_1$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_3$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_4 \to \tau_5$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_4 \to \tau_6 \to \tau_7$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_4 \to \tau_6 \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_8 \to \tau_6 \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_8 \to \tau_6 \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : (\tau_{10} \to \tau_{11}) \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : (\tau_{10} \to \tau_8 \to \tau_9) \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Bonus: použití STLC

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$

Axiom	Odvození	Substituce
–	$\vdash (\lambda fxy.(fy)x) : \tau_1$	–
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Curry-Howardova korespondence

Co vlastně znamená typ nějakého výrazu?

- Tradiční jazyky: Formát dat (zhruba)
- Funkcionální programování: jaký formát má $\rightarrow$?

Motivace

Motivační úkol: Napište funkci f, která má typ $f :: a \to b$.

Motivace

Motivační úkol: Napište funkci f, která má typ $f :: a \to b$.

1. Dostanu a, zahodím ho, OK.

Motivace

Motivační úkol: Napište funkci f, která má typ $f :: a \to b$.

1. Dostanu a, zahodím ho, OK.
2. Vyrobím nějaké b?

Motivace

Motivační úkol: Napište funkci f, která má typ $f :: a \to b$.

1. Dostanu a, zahodím ho, OK.
2. Vyrobím nějaké b?
   $(\lambda x \to 1) :: a \to \mathit{Integer} \not\equiv a \to b$

$f :: a \to b$ znamená $f :: \forall a.\forall b.a \to b$. Program tímto slibuje, že bude umět vyrobit jakékoliv b.

Motivace

Motivační úkol: Napište funkci f, která má typ $f :: a \to b$.

1. Dostanu a, zahodím ho, OK.
2. Vyrobím nějaké b?
   $(\lambda x \to 1) :: a \to \mathit{Integer} \not\equiv a \to b$

$f :: a \to b$ znamená $f :: \forall a.\forall b.a \to b$. Program tímto slibuje, že bude umět vyrobit jakékoliv b.

Haskell-style řešení:

$$(\lambda\_ \to \bot) :: p \to a$$

Sémantika $\rightarrow$

Použitím $a \rightarrow b$ tvrdíme, že z věci s typem a umíme vyrobit věc s typem b.

Sémantika $\rightarrow$

Použitím $a \rightarrow b$ tvrdíme, že z věci s typem a umíme vyrobit věc s typem b.

Typy jako tvrzení

- $\rightarrow$ si můžeme představit jako logickou implikaci
- místo konjunkce $a \wedge b$ použijeme typ n-tice (a, b)
- místo disjunkce $a \vee b$ použijeme *Either a b*

Např.: viditelně neplatí:

$$a \rightarrow (a, b) \qquad a \implies (a \wedge b)$$

ale pokud nám někdo navíc dá způsob jak vyrábět b z a, může platit:

$$(a \rightarrow b) \rightarrow (a \rightarrow (a, b)) \qquad (a \implies b) \implies (a \implies (a \wedge b))$$

Curry-Howard: Program s typem t existuje právě, když je tvrzení odpovídající t dokazatelné v přirozené logice.

Programy jako důkazy

$$(\lambda x \to (x, \bot)) :: a \to (a, b)$$

(program obsahuje chybu, tvrzení není dokázané)

$$(\lambda f\ x \to (x, f\ x)) :: (a \to b) \to (a \to (a, b))$$

(tvrzení je dokázané)

Korespondence

Axiom logiky	λ-kalkulus
Zeslabení	$\vdash K : \alpha \to \beta \to \alpha$
$(\phi \implies (\psi \implies \phi))$	
Substituce	$S = (\lambda xyz.xz(yz))$
	$\vdash S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
Modus ponens	$\{A : \alpha \to \beta, B : \alpha\} \vdash AB : \beta$
	$\vdash I : (\alpha \to \beta) \to \alpha \to \beta$
Hledání důkazu	Inhabitace typu

Extra korespondující tvrzení:

- Kombinováním axiomů lze sestrojit jakékoliv tvrzení
- Kombinováním *SKI* lze sestrojit jakýkoliv program (včetně Turingova stroje)

Korespondence

Axiom logiky	λ-kalkulus
Zeslabení	$\vdash K : \alpha \to \beta \to \alpha$
$(\phi \implies (\psi \implies \phi))$	
Substituce	$S = (\lambda xyz.xz(yz))$
	$\vdash S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
Modus ponens	$\{A : \alpha \to \beta, B : \alpha\} \vdash AB : \beta$
	$\vdash I : (\alpha \to \beta) \to \alpha \to \beta$
Hledání důkazu	Inhabitace typu

Extra korespondující tvrzení:

- Kombinováním axiomů lze sestrojit jakékoliv tvrzení
- Kombinováním *SKI* lze sestrojit jakýkoliv program (včetně Turingova stroje)

Příklad

Chci dokázat: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

Příklad

Chci dokázat: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

$$p :: (a, \mathit{Either}\ b\ c) \to \mathit{Either}\ (a, c)\ (a, b)$$

Příklad

Chci dokázat: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

$$p :: (a, \mathit{Either}\ b\ c) \to \mathit{Either}\ (a, c)\ (a, b)$$

$$p\ (a, e) = \textbf{case}\ e\ \textbf{of}\ \mathit{Left}\ b \to \mathit{Right}\ (a, b)$$
$$\mathit{Right}\ c \to \mathit{Left}\ (a, c)$$

Výhoda: prohledávání termů je algoritmicky poměrně jednoduché.

Hlavní implementace: Agda, Coq, HOL, Djinn

Co s negací?

Logika	λ-kalkulus
True	$()$
False	*Void*
$\neg a$	$a \to \mathit{Void}$
Spornost teorie	$\bot$

Void nejde vyrobit, můžeme ho jen ‘přeposlat’ dál.

Pozor, v intuicionistické logice neplatí $\neg\neg a = a$!

Co s negací?

Logika	λ-kalkulus
True	$()$
False	Void
$\neg a$	$a \to \mathit{Void}$
Spornost teorie	$\bot$

Void nejde vyrobit, můžeme ho jen ‘přeposlat’ dál.

Pozor, v intuicionistické logice neplatí $\neg\neg a = a$!

S trochou přidané magie jde vyrobit:

$$\mathit{throw} :: a \to \mathit{Void}$$
$$\mathit{catch} :: ((a \to \mathit{Void}) \to \mathit{Void}) \to a$$

Co si z toho odnést?

- Funkce je důkazem existence datové transformace
- Kód funkce jde automaticky odvodit z typu
  - přesnější typ $\Longrightarrow$ méně validních implementací
    $\Longrightarrow$ méně chyb
  - implementaci podle dostatečně přesného typu může zařídit počítač (viz. Coq&Agda)
  - obecnější typ může být restriktivnější
- Je vhodné se podobně chovat i ke specifikacím dat
  (data mají typ $\Longleftrightarrow$ jsou validní)

Co si z toho odnést?

- Funkce je důkazem existence datové transformace
- Kód funkce jde automaticky odvodit z typu
  - přesnější typ $\Longrightarrow$ méně validních implementací
    $\Longrightarrow$ méně chyb
  - implementaci podle dostatečně přesného typu může zařídit počítač (viz. Coq&Agda)
  - obecnější typ může být restriktivnější
- Je vhodné se podobně chovat i ke specifikacím dat
  (data mají typ $\Longleftrightarrow$ jsou validní)

Těsné typy

Jak produkovat typově odolný kód:

1. psát generičtější funkce
2. používat specifičtější datové typy (přesnější, ‘těsné’, ‘tight’)

Generické funkce fungují na víc dat (šetří kód), jejich typ je ale ve skutečnosti restriktivnější!

Specifičtější data výrazně zjednodušují okolní program (je v něm míň ifů).

Těsné typy

Typické nepřesné datové konstrukce:

- *null*-member místo více variant celého objektu
- seznamy místo *n*-tic nebo záznamů
- XML

Těsné typy

Opačný problém: Kompilátor CakeML má (přesné) typy pro všech cca. 20 různých reprezentací mezikódu v pipeline.

- Výhoda: verifikovat ho je zábava na 10 minut
- Nevýhoda: Není rozumné psát skoro stejnou funkci pro 20 různých datových typů
- Řešení:

Těsné typy

Opačný problém: Kompilátor CakeML má (přesné) typy pro všech cca. 20 různých reprezentací mezikódu v pipeline.

- Výhoda: verifikovat ho je zábava na 10 minut
- Nevýhoda: Není rozumné psát skoro stejnou funkci pro 20 různých datových typů
- Řešení: feature-based typové třídy poskytující generický přístup ke specifickým vlastnostem.
  - Např. *HasVars*, *Formattable*, ...
  - THIH:
    - *Type*, *Pred*, *Qual t*, *Scheme*, *Assump* $\in$ *Types*
    - *Type*, [α], *Pred*, *Qual t* $\in$ *Instantiate*
  - -XOverloadedRecords

Optika, čočky a hranoly

OOP *universe* · *galaxy* (*milky*) · *sun* · *earth* · *matfyz* · *rooms* [*S4*] · *molecules* [23] · *atom* [0] $\leadsto$ `"N"`

Haskell (*head* · *atoms* · (!!23) · *molecules* · (!!*S4*) · *rooms* · *matfyz* · *earth* · *sun* · (!!*milky*) · *galaxies*) *universe* $\leadsto$ `"N"`

Motivace

OOP $\mathit{universe} \cdot \mathit{galaxy}\ (\mathit{milky}) \cdot \mathit{sun} \cdot \mathit{earth} \cdot \mathit{matfyz} \cdot \mathit{rooms}\ [\mathit{S4}] \cdot \mathit{molecules}\ [23] \cdot \mathit{atom}\ [0] \quad \rightsquigarrow \texttt{"N"}$

Haskell $(\mathit{head} \cdot \mathit{atoms} \cdot (!!23) \cdot \mathit{molecules} \cdot (!!\mathit{S4}) \cdot \mathit{rooms} \cdot \mathit{matfyz} \cdot \mathit{earth} \cdot \mathit{sun} \cdot (!!\mathit{milky}) \cdot \mathit{galaxies})\ \mathit{universe} \quad \rightsquigarrow \texttt{"N"}$

OOP $\mathit{universe} \cdot \mathit{galaxy}\ (\mathit{milky}) \cdot \mathit{sun} \cdot \mathit{earth} \cdot \mathit{matfyz} \cdot \mathit{rooms}\ [\mathit{S4}] \cdot \mathit{molecules}\ [23] \cdot \mathit{atom}\ [1] = \texttt{"Au"}$

Haskell ☹

Problém

Vybírání podstruktur v OOP ve skutečnosti nevrací data, ale *reference*. Haskell jednoduché jazykové reference nemá. (*IORef* apod. se nepočítá)

Co s tím?

Problém

Vybírání podstruktur v OOP ve skutečnosti nevrací data, ale *reference*. Haskell jednoduché jazykové reference nemá. (*IORef* apod. se nepočítá)

Co s tím? Software engineering! Reference nasimulujeme.

Co jde dělat s referencí?

- get
- set
- vyrobit z ní subreferenci

Jaká funkcionální hodnota může mít všechny tyto vlastnosti najednou?

Co jde dělat s referencí? (přesněji)

- vybrat hluboce zanořenou část nějakého velkého objektu
- změnit část nějakého objektu
- skombinovat ji s jinou (sub)referencí a dostat hlubší referenci

Co jde dělat s referencí? (přesněji)

- vybrat hluboce zanořenou část nějakého velkého objektu
- změnit část nějakého objektu
- skombinovat ji s jinou (sub)referencí a dostat hlubší referenci

První pokus:

$$\begin{aligned}&\textbf{data}\ \mathit{Ref}\ \mathit{big}\ \mathit{small} = \mathit{Ref}\\&\quad \{\mathit{view} :: \mathit{big} \to \mathit{small}\\&\quad ,\mathit{over} :: (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{big})\\&\quad \}\end{aligned}$$

$$\textbf{data}\ \mathit{Ref}\ \mathit{big}\ \mathit{small} = \mathit{Ref}$$
$$\quad \{\mathit{view} :: \mathit{big} \to \mathit{small}, \mathit{over} :: (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{big})\}$$

$$\textbf{data}\ \mathit{Dva}\ a = \mathit{Dva}\ a\ a\ \textbf{deriving}\ \mathit{Show}$$

$$x = \mathit{Ref}\ (\lambda(\mathit{Dva}\ a\ \_) \to a)\ (\lambda f\ (\mathit{Dva}\ a\ b) \to \mathit{Dva}\ (f\ a)\ b)$$
$$y = \mathit{Ref}\ (\lambda(\mathit{Dva}\ \_\ b) \to b)\ (\lambda f\ (\mathit{Dva}\ a\ b) \to \mathit{Dva}\ a\ (f\ b))$$

$$t = \mathit{Dva}\ 1\ 2$$

$$\mathit{view}\ x\ t \quad \rightsquigarrow 1$$
$$\mathit{over}\ y\ (+1)\ t \quad \rightsquigarrow \mathit{Dva}\ 1\ 3$$

$$\mathit{set}\ x\ v = \mathit{over}\ x\ (\mathit{const}\ v)$$
$$\mathit{set}\ x\ 5\ t \quad \rightsquigarrow \mathit{Dva}\ 5\ 2$$

$$(\mathit{Ref}\ \mathit{v1}\ \mathit{o1})\ `\mathit{sub}`\ (\mathit{Ref}\ \mathit{v2}\ \mathit{o2}) = \mathit{Ref}\ (\mathit{v2} \cdot \mathit{v1})\ (\mathit{o1} \cdot \mathit{o2})$$

$$t = \mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4)$$

$$: t\ x\ `\mathit{sub}`\ y \quad \rightsquigarrow \mathit{Ref}\ (\mathit{Dva}\ (\mathit{Dva}\ b))\ b$$

$$\mathit{view}\ (x\ `\mathit{sub}`\ y)\ t \quad \rightsquigarrow 2$$

$$\mathit{over}\ (x\ `\mathit{sub}`\ y)\ (*10)\ t \quad \rightsquigarrow \mathit{Dva}\ (\mathit{Dva}\ 1\ 20)\ (\mathit{Dva}\ 3\ 4)$$

$$\mathit{set}\ r\ x = \mathit{over}\ r\ (\mathit{const}\ x)$$

$$\mathit{set}\ (x\ `\mathit{sub}`\ y)\ 1234\ t \quad \rightsquigarrow \mathit{Dva}\ (\mathit{Dva}\ 1\ 1234)\ (\mathit{Dva}\ 3\ 4)$$

Zlepšení

- Všichni jsou zvyklí na tečku! Chceme $x \cdot y \cdot x$
- Co takhle něco lepšího než tradiční OOP?
  - Jak správně udělat referenci na objekt, který nemusí být k dispozici?
    left :: *Ref* (*Either a b*) *a* ?
  - Nejde udělat referenci na víc objektů, podobně ve vektorových jazycích?
    Clamp v Rku: $x\ [x < 0] = 0; x\ [x > 1] = 1;$
  - Co když chceme změnit typ uvnitř velkého objektu?

Zlepšení

- Všichni jsou zvyklí na tečku! Chceme $x \cdot y \cdot x$
- Co takhle něco lepšího než tradiční OOP?
  - Jak správně udělat referenci na objekt, který nemusí být k dispozici?
    left :: *Ref* (*Either a b*) *a* ?
  - Nejde udělat referenci na víc objektů, podobně ve vektorových jazycích?
    Clamp v Rku: $x\ [x < 0] = 0; x\ [x > 1] = 1;$
  - Co když chceme změnit typ uvnitř velkého objektu?

Začneme tím složitějším.

Pozn.: Protože *Ref big small* reprezentuje ‘zvětšení’ operace na malém objektu, říká se jí *Lens*.

Optika, pokus 2

Máme:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$

Chceme:

$$(\cdot) :: \mathit{Lens}\ x\ y \to \mathit{Lens}\ y\ z \to \mathit{Lens}\ x\ z$$

Lens tedy musí být funkce:

$$\textbf{type}\ \mathit{Lens}\ a\ b = \_ \to \_$$

Zároveň víme:

$$\mathit{set}' :: (\_ \to \_) \to (b \to b) \to (a \to a)$$
$$\mathit{get}' :: (\_ \to \_) \to (b \to b) \to (a \to b)$$

Z jakého $(\_ \to \_)$ jde jednoduše udělat *get* i *set*?

Optika, pokus 2

Pokračujeme:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(\cdot) :: \mathit{Lens}\ x\ y \to \mathit{Lens}\ y\ z \to \mathit{Lens}\ x\ z$$

Pořadí parametrů musí odpovídat

$$\mathit{Lens}\ \mathit{big}\ \mathit{small} = \mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}$$

tj. čočka jako funkce konvertuje ‘Něco pracující s částí’ na ’Něco pracující s celkem’.

Optika, pokus 2

Pokračujeme:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(\cdot) :: \mathit{Lens}\ x\ y \to \mathit{Lens}\ y\ z \to \mathit{Lens}\ x\ z$$

Pořadí parametrů musí odpovídat

$$\mathit{Lens}\ \mathit{big}\ \mathit{small} = \mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}$$

tj. čočka jako funkce konvertuje 'Něco pracující s částí' na 'Něco pracující s celkem'.

Takovou funkci potřebujeme použít v:

$$\mathit{set}' :: (\mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{big})$$
$$\mathit{get}' :: (\mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{small})$$

Optika, pokus 2

Pokračujeme:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(\cdot) :: \mathit{Lens}\ x\ y \to \mathit{Lens}\ y\ z \to \mathit{Lens}\ x\ z$$

Pořadí parametrů musí odpovídat

$$\mathit{Lens}\ \mathit{big}\ \mathit{small} = \mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}$$

tj. čočka jako funkce konvertuje 'Něco pracující s částí' na 'Něco pracující s celkem'.

Takovou funkci potřebujeme použít v:

$$\mathit{set}' :: (\mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{big})$$
$$\mathit{get}' :: (\mathit{Neco}\ \mathit{small} \to \mathit{Neco}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{small})$$

Neco big očividně musí být konvertovatelné na *small* i *big*!

Optika, pokus 2

Za cenu drobného zkomplikování interface můžeme dovnitř vrazit nějakou (parametrizovatelnou) dekoraci:

$$\textbf{type}\ \mathit{Neco}\ a = a \to \mathit{Dekorace}\ a$$

$$\mathit{set}' :: ((\mathit{small} \to \mathit{Set\ small}) \to (\mathit{big} \to \mathit{Set\ big})) \to (\mathit{small} \to \mathit{Set\ small}) \to (\mathit{big} \to \mathit{Set\ big})$$
$$\mathit{get}' :: ((\mathit{small} \to \mathit{Get\ small}) \to (\mathit{big} \to \mathit{Get\ big})) \to (\mathit{small} \to \mathit{Get\ small}) \to (\mathit{big} \to \mathit{Get\ big})$$

Uživatel dodá funkce pracující s dekorací, tu si nakonec sám odstraní.

Optika, pokus 2

Za cenu drobného zkomplikování interface můžeme dovnitř vrazit nějakou (parametrizovatelnou) dekoraci:

$$\textbf{type}\ \mathit{Neco}\ a = a \to \mathit{Dekorace}\ a$$

$$\mathit{set}' :: ((\mathit{small} \to \mathit{Set}\ \mathit{small}) \to (\mathit{big} \to \mathit{Set}\ \mathit{big})) \to (\mathit{small} \to \mathit{Set}\ \mathit{small}) \to (\mathit{big} \to \mathit{Set}\ \mathit{big})$$
$$\mathit{get}' :: ((\mathit{small} \to \mathit{Get}\ \mathit{small}) \to (\mathit{big} \to \mathit{Get}\ \mathit{big})) \to (\mathit{small} \to \mathit{Get}\ \mathit{small}) \to (\mathit{big} \to \mathit{Get}\ \mathit{big})$$

Uživatel dodá funkce pracující s dekorací, tu si nakonec sám odstraní. Potřebujeme, aby po odstranění dekorace platilo:

$$\begin{aligned} \mathit{Set}\ \mathit{small} &\rightsquigarrow \mathit{small} \\ \mathit{Set}\ \mathit{big} &\rightsquigarrow \mathit{big} \\ \mathit{Get}\ \mathit{small} &\rightsquigarrow \mathit{small} \\ \mathit{Get}\ \mathit{big} &\rightsquigarrow \mathit{small} \end{aligned}$$

Optika, pokus 2

Za cenu drobného zkomplikování interface můžeme dovnitř vrazit nějakou (parametrizovatelnou) dekoraci:

type *Neco a* = *a* → *Dekorace a*

set′ :: ((*small* → *Set small*) → (*big* → *Set big*)) → (*small* → *Set small*) → (*big* → *Set big*)
get′ :: ((*small* → *Get small*) → (*big* → *Get big*)) → (*small* → *Get small*) → (*big* → *Get big*)

Uživatel dodá funkce pracující s dekorací, tu si nakonec sám odstraní. Potřebujeme, aby po odstranění dekorace platilo:

Set small	⇝ *small*		*id*	*S*	⇝ *S*
Set big	⇝ *big*		*id*	*B*	⇝ *B*
Get small	⇝ *small*		*const S S*		⇝ *S*
Get big	⇝ *small*		*const S B*		⇝ *S*


Vorsicht
Funktor
2 m Abstand halten

Optika, pokus 2

$$
\begin{aligned}
&\mathit{runIdentity} :: \mathit{Identity\ small} \to \mathit{small}\\
&\mathit{runIdentity} :: \mathit{Identity\ big} \to \mathit{big}\\
&\mathit{getConst} :: (\mathit{Const\ small})\ \mathit{small} \to \mathit{small}\\
&\mathit{getConst} :: (\mathit{Const\ small})\ \mathit{big} \to \mathit{small}
\end{aligned}
$$

Dostáváme:

$$
\begin{aligned}
&\mathit{set}' :: ((\mathit{small} \to \mathit{Identity\ small}) \to (\mathit{big} \to \mathit{Identity\ big})) \to (\mathit{small} \to \mathit{Identity\ small}) \to (\mathit{big} \to \mathit{Identity\ big})\\
&\mathit{set}' = \mathit{id}\\
&\mathit{get}' :: ((\mathit{small} \to \mathit{Const\ small\ small}) \to (\mathit{big} \to \mathit{Const\ small\ big})) \to (\mathit{small} \to \mathit{Const\ small\ small}) \to (\mathit{big} \to \mathit{Const\ small\ big})\\
&\mathit{get}' = \mathit{id}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{over}\ \mathit{lens}\ f = \mathit{runIdentity} \cdot \mathit{lens}\ (\mathit{Identity} \cdot f)\\
&\mathit{view}\ \mathit{lens}\ f = \mathit{getConst} \cdot \mathit{lens}\ (\mathit{Const} \cdot f)
\end{aligned}
$$

Co se zlepšilo?

Naivní reference:

$$\textbf{data}\ \mathit{Ref}\ b\ s = \{\mathit{view} :: b \to s,\ \mathit{over} :: (s \to s) \to (b \to b)\}$$

Problémy:

- Co když někdo vymyslí třetí operaci?
- Co když chceme referovat na víc prvků?
- Redundantní implementace ‘vybrání prvku’ ve *view* i *over*

Funktorová reference:

$$\mathit{Lens}\ b\ s :: \forall f.\mathit{Functor}\ f \Rightarrow (s \to f\ s) \to b \to f\ b$$

- Implementuje jedinou operaci $b \to (s, s \to b)$
- Pipeline: $b \to s \to f\ s \to f\ b$
- Konkrétní operace se vybere pomocí dodaného funktoru
  $\mathit{Identity}\ b \equiv b$, $(\mathit{Const}\ s)\ b \equiv s$
- Další operace jde implementovat libovolným dalším funktorem

Optika — implementace

Jak vypadá implementace čočky?

$$\mathit{Lens\ big\ small} :: \forall f.\mathit{Functor}\ f \Rightarrow (\mathit{small} \to f\ \mathit{small}) \to \mathit{big} \to f\ \mathit{big}$$

- dostane funkci, která malou ‘součást’ něčím obalí
- má vyrobit funkci, která umí tím samým obalit velký ‘celek’
- moc typově validních implementací nemáme:
  1. celek dostaneme (jako parametr typu *big*)
  2. z celku vyřízneme část typu *small*
  3. *small* si můžeme nechat obalit na $f\ \mathit{small}$
  4. z $f\ \mathit{small}$ chceme udělat $f\ \mathit{big}$, ale umíme jen $\mathit{small} \to \mathit{big}$
  5. když je f funktor, můžeme to udělat pomocí *fmap*

Lens jde takto triviálně vyrobit z getteru a setteru.

Optika — implementace

Jak vypadá implementace čočky?

$$\mathit{Lens\ big\ small} :: \forall f.\mathit{Functor}\ f \Rightarrow (\mathit{small} \to f\ \mathit{small}) \to \mathit{big} \to f\ \mathit{big}$$

- dostane funkci, která malou ‘součást’ něčím obalí
- má vyrobit funkci, která umí tím samým obalit velký ‘celek’
- moc typově validních implementací nemáme:
  1. celek dostaneme (jako parametr typu *big*)
  2. z celku vyřízneme část typu *small*
  3. *small* si můžeme nechat obalit na *f small*
  4. z *f small* chceme udělat *f big*, ale umíme jen *small* → *big*
  5. když je *f* funktor, můžeme to udělat pomocí *fmap*

Lens jde takto triviálně vyrobit z getteru a setteru.

Optika — implementace

Jak vypadá implementace čočky?

$$\mathit{Lens\ big\ small} :: \forall f.\mathit{Functor}\ f \Rightarrow (\mathit{small} \to f\ \mathit{small}) \to \mathit{big} \to f\ \mathit{big}$$

- dostane funkci, která malou ‘součást’ něčím obalí
- má vyrobit funkci, která umí tím samým obalit velký ‘celek’
- moc typově validních implementací nemáme:
  1. celek dostaneme (jako parametr typu *big*)
  2. z celku vyřízneme část typu *small*
  3. *small* si můžeme nechat obalit na $f\ \mathit{small}$
  4. z $f\ \mathit{small}$ chceme udělat $f\ \mathit{big}$, ale umíme jen $\mathit{small} \to \mathit{big}$
  5. když je f funktor, můžeme to udělat pomocí *fmap*

Lens jde takto triviálně vyrobit z getteru a setteru.

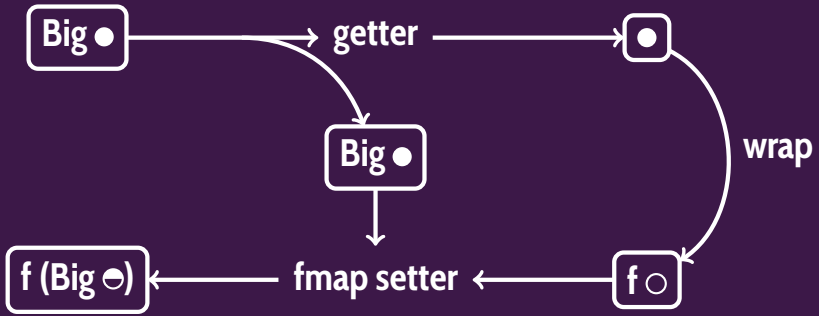
Optika — implementace

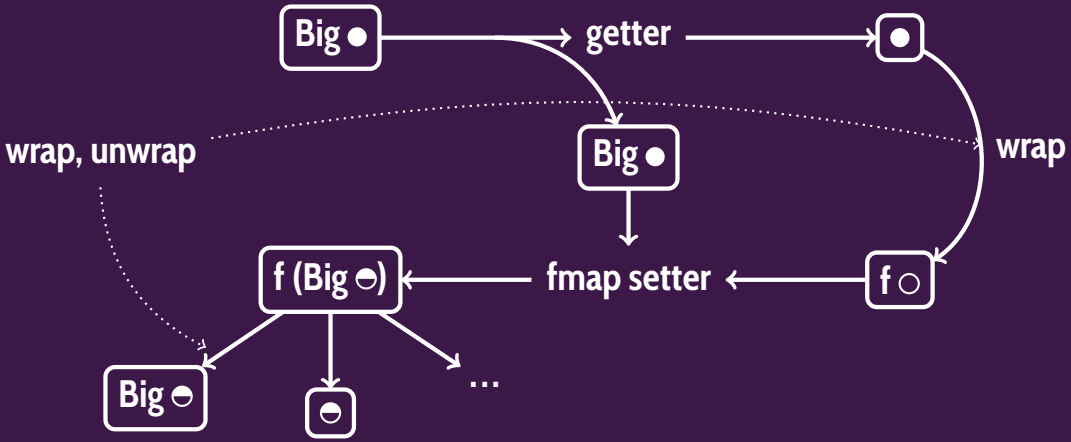
Jak vypadá implementace čočky?

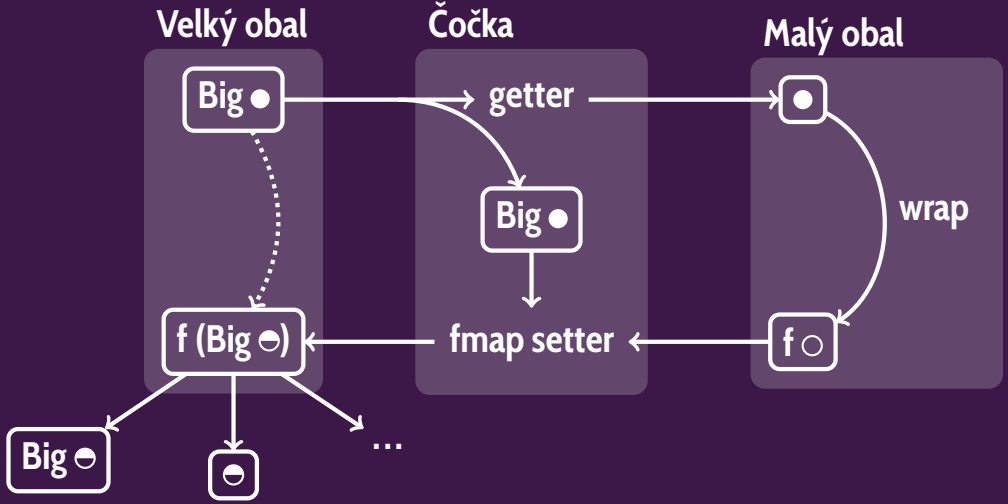
$$\mathit{Lens\ big\ small} :: \forall f.\mathit{Functor}\ f \Rightarrow (\mathit{small} \to f\ \mathit{small}) \to \mathit{big} \to f\ \mathit{big}$$

- dostane funkci, která malou ‘součást’ něčím obalí
- má vyrobit funkci, která umí tím samým obalit velký ‘celek’
- moc typově validních implementací nemáme:
  1. celek dostaneme (jako parametr typu *big*)
  2. z celku vyřízneme část typu *small*
  3. *small* si můžeme nechat obalit na $f\ \mathit{small}$
  4. z $f\ \mathit{small}$ chceme udělat $f\ \mathit{big}$, ale umíme jen $\mathit{small} \to \mathit{big}$
  5. když je f funktor, můžeme to udělat pomocí *fmap*

Lens jde takto triviálně vyrobit z getteru a setteru.


Big ●
getter
●
wrap
Big ●
f (Big ◓)
fmap setter
f ○


Big ●
getter
●
wrap, unwrap
Big ●
wrap
f (Big ◓)
fmap setter
f ○
Big ◓
◓
…


Velký obal
Čočka
Malý obal
Big ●
getter
●
Big ●
wrap
f (Big ◓)
fmap setter
f ○
Big ◓
◓
...

Optika – výsledek

Celkový typ čočky (potřebuje `-XRankNTypes`):

$$\textbf{type}\ \mathit{Lens}\ a\ b = \forall f.\mathit{Functor}\ f \Rightarrow (b \to f\ b) \to (a \to f\ a)$$

Jak vypadá funktorový lens?

$$x :: \mathit{Lens}\ (\mathit{Dva}\ a)\ a$$
$$x :: \mathit{Functor}\ f \Rightarrow (a \to f\ a) \to (\mathit{Dva}\ a \to f\ (\mathit{Dva}\ a))$$
$$x\ \mathit{wrap}\ (\mathit{Dva}\ a\ b) = \mathit{fmap}\ (\lambda a' \to \mathit{Dva}\ a'\ b)\ \$\ \mathit{wrap}\ a$$
$$y\ \mathit{wrap}\ (\mathit{Dva}\ a\ b) = \mathit{fmap}\ (\lambda b' \to \mathit{Dva}\ a\ b')\ \$\ \mathit{wrap}\ b$$

Genericky z getteru a setteru:

$$\mathit{lens} :: (a \to b) \to (a \to b \to a) \to \mathit{Lens}\ a\ b$$
$$\mathit{lens}\ \mathit{getter}\ \mathit{setter}\ \mathit{wrap}\ a = \mathit{fmap}\ (\mathit{setter}\ a)\ \$\ \mathit{wrap}\ (\mathit{getter}\ a)$$

Optika vs. tečka

Typový úspěch:

$$
\begin{aligned}
(\cdot) :: & \mathit{Functor}\ f \\
& \Rightarrow ((b \to f\ b) \to (a \to f\ a)) \\
& \to ((c \to f\ c) \to (b \to f\ b)) \\
& \to ((c \to f\ c) \to (a \to f\ a))
\end{aligned}
$$

$$(\cdot) :: \mathit{Lens}\ a\ b \to \mathit{Lens}\ b\ c \to \mathit{Lens}\ a\ c$$

(a, b jsou v definici *Lens* prohozené!)

Optika — implementace

Tradičně k dispozici:

$$
\begin{aligned}
&\mathit{set} :: \mathit{Lens}\ a\ b \to b \to a \to a \\
&\mathit{set}\ l\ v = \mathit{runIdentity} \cdot l\ (\mathit{Identity} \cdot \mathit{const}\ v) \\
&\mathit{over} :: \mathit{Lens}\ a\ b \to (b \to b) \to a \to a \\
&\mathit{over}\ l\ f = \mathit{runIdentity} \cdot l\ (\mathit{Identity} \cdot f) \\
&\mathit{view} :: \mathit{Lens}\ a\ b \to a \to b \\
&\mathit{view}\ l = \mathit{getConst} \cdot l\ \mathit{Const}
\end{aligned}
$$

Optika — použití

$$\mathit{set}\ x :: b \to \mathit{Dva}\ b \to \mathit{Dva}\ b$$
$$\mathit{over}\ (x \cdot y)\ (+1) :: \mathit{Num}\ b \Rightarrow \mathit{Dva}\ (\mathit{Dva}\ b) \to \mathit{Dva}\ (\mathit{Dva}\ b)$$
$$\mathit{view}\ (x \cdot y \cdot x) :: \mathit{Dva}\ (\mathit{Dva}\ (\mathit{Dva}\ b)) \to b$$

Příjemný bonus: Lensy se skládají v přirozenějším pořadí

$$\begin{aligned} &\mathit{set}\ (x \cdot y)\ 10\ (\mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4)) \\ \leadsto\quad &\mathit{Dva}\ (\mathit{Dva}\ 1\ 10)\ (\mathit{Dva}\ 3\ 4) \end{aligned}$$

Optika — použití

$$\mathit{set}\ x :: b \to \mathit{Dva}\ b \to \mathit{Dva}\ b$$
$$\mathit{over}\ (x \cdot y)\ (+1) :: \mathit{Num}\ b \Rightarrow \mathit{Dva}\ (\mathit{Dva}\ b) \to \mathit{Dva}\ (\mathit{Dva}\ b)$$
$$\mathit{view}\ (x \cdot y \cdot x) :: \mathit{Dva}\ (\mathit{Dva}\ (\mathit{Dva}\ b)) \to b$$

Příjemný bonus: Lensy se skládají v přirozenějším pořadí

$$\begin{aligned} &\mathit{set}\ (x \cdot y)\ 10\ (\mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4)) \\ \rightsquigarrow\quad &\mathit{Dva}\ (\mathit{Dva}\ 1\ 10)\ (\mathit{Dva}\ 3\ 4) \end{aligned}$$

Původce: Twan van Laarhoven (2009)

✎ `https://www.twanvl.nl/blog/haskell/cps-functional-references`

Edward Kmett (import Control.Lens)

"**newtype** $PlanT\ k\ i\ o\ m\ a = PlanT\ \{runPlanT :: \forall r.(a \to m\ r) \to (o \to m\ r \to m\ r) \to (\forall z.(z \to m\ r) \to k\ i\ z \to m\ r \to m\ r) \to m\ r \to m\ r\}$ **is a monad I actually use.**"

Víc optiky!

Problémy:

- *allItems* :: *Lens* $[a]$ _
- *left* :: *Lens* (*Either a b*) _

Víc optiky!

Problémy:

- *allItems* :: *Lens* [*a*] _
- *left* :: *Lens* (*Either a b*) _

Konceptům se říká Traversal a Prism.

Pro implementaci pokročilých čoček je (občas) potřeba použít ještě generičtější typy. Intuice z jednoduchého funktorového *Lens* ale platí.

Víc optiky!

Problémy:

- *allItems* :: *Lens* $[a]$ _
- *left* :: *Lens* $(Either\ a\ b)$ _

Konceptům se říká Traversal a Prism.

Pro implementaci pokročilých čoček je (občas) potřeba použít ještě generičtější typy. Intuice z jednoduchého funktorového *Lens* ale platí.

Traversal

Pokud chceme optiku aplikovat na víc věcí, musí funktor podporovat možnost spojovat obaly.
Tradičním řešením je *Applicative*:

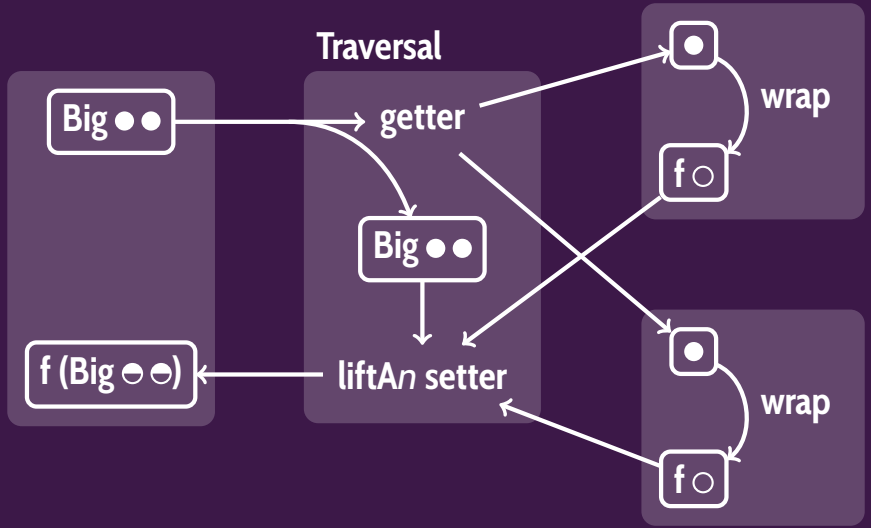
$$\textbf{data}\ \mathit{Dva}\ a = \mathit{Dva}\ a\ a\ \textbf{deriving}\ \mathit{Show}$$

$$\mathit{oba}\ \mathit{ff}\ (\mathit{Dva}\ a\ b) = \mathit{Dva}\ \langle\$\rangle\ \mathit{ff}\ a\ \langle*\rangle\ \mathit{ff}\ b$$

Výsledky vypadají zajímavě:

$$a = \mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4)$$

$$\begin{aligned}
&\mathit{set}\ \mathit{oba}\ 5\ a && \rightsquigarrow \mathit{Dva}\ 5\ 5\\
&\mathit{set}\ (\mathit{oba} \cdot x)\ 5\ a && \rightsquigarrow \mathit{Dva}\ (\mathit{Dva}\ 5\ 2)\ (\mathit{Dva}\ 5\ 4)\\
&\mathit{set}\ (x \cdot \mathit{oba})\ 5\ a && \rightsquigarrow \mathit{Dva}\ (\mathit{Dva}\ 5\ 5)\ (\mathit{Dva}\ 3\ 4)\\
&\mathit{over}\ (\mathit{oba} \cdot y)\ (+10)\ a && \rightsquigarrow \mathit{Dva}\ (\mathit{Dva}\ 1\ 12)\ (\mathit{Dva}\ 3\ 14)
\end{aligned}$$


Traversal
Big ●●
getter
wrap
f ○
Big ●●
f (Big ◒ ◒)
liftA*n* setter
wrap
f ○

```
:t view oba
```

```
:t view oba
```

$$view\ oba :: Monoid\ c \Rightarrow Dva\ c \rightarrow c$$

kde se tady vzal monoid?

view oba :: *Monoid c* $\Rightarrow$ *Dva c* $\rightarrow$ *c*

Důvod je v definici aplikativní instance *Const*:

instance *Monoid b* $\Rightarrow$ *Applicative* (*Const b*)

Použití:

view oba (*Dva* (*Sum* 1) (*Sum* 2))
$\rightsquigarrow$ *Sum* $\{$*getSum* $= 3\}$

view oba (*Dva* $[1, 2]$ $[3, 4]$)
$\rightsquigarrow$ $[1, 2, 3, 4]$

Traverzování

Překvapivě, *traverse* je traversal pro všechno traverzovatelné:

$$\begin{aligned}&\mathit{traverse} :: (\mathit{Applicative}\ f, \mathit{Traversable}\ t) \Rightarrow \\ &\quad (a \to f\ b) \to (t\ a \to f\ (t\ b))\end{aligned}$$

(typ je jen o trochu košatější, než u *Lens a b*)

$$\begin{aligned}&\mathit{over}\ (\mathit{traverse} \cdot y)\ (+1)\ [\mathit{Dva}\ 1\ 2, \mathit{Dva}\ 3\ 4] \\ &\qquad \rightsquigarrow [\mathit{Dva}\ 1\ 3, \mathit{Dva}\ 3\ 5]\end{aligned}$$

$$\mathit{listOf}\ l = \mathit{getConst} \cdot l\ (\lambda x \to \mathit{Const}\ [x])$$

$$\begin{aligned}&\mathit{listOf}\ \mathit{oba}\ \$\ \mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4) \\ &\qquad \rightsquigarrow [\mathit{Dva}\ 1\ 2, \mathit{Dva}\ 3\ 4]\end{aligned}$$

$$\mathit{listOf}\ (\mathit{oba} \cdot y)\ \$\ \mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4) \quad \rightsquigarrow [2, 4]$$

$$\mathit{listOf}\ (y \cdot \mathit{oba})\ \$\ \mathit{Dva}\ (\mathit{Dva}\ 1\ 2)\ (\mathit{Dva}\ 3\ 4) \quad \rightsquigarrow [3, 4]$$

SQL cvičení:

```
ident    ff s = ff    s   -- nic nedělá
ignored ff s = pure s    -- zabrání změnám
filtered cond f s = if cond s then f s else pure s
over (oba · oba · filtered even) (`div`2) $ Dva (Dva 1 2) (Dva 3 4)
      ⇝ Dva (Dva 1 1) (Dva 3 2)
```

Iso

Iso je čočka měnící reprezentaci (název podle izomorfismu).

$$\begin{aligned}
&\mathit{asString} :: (\mathit{Show}\ a, \mathit{Read}\ a) \Rightarrow \mathit{Lens}\ a\ \mathit{String} \\
&\mathit{asString}\ \mathit{ff}\ a = \mathit{fmap}\ \mathit{read}\ \$\ \mathit{ff}\ (\mathit{show}\ a) \\
&\mathit{set}\ (\mathit{oba} \cdot \mathit{asString} \cdot \mathit{traverse} \cdot \mathit{filtered}\ (\equiv \texttt{'3'}))\ \texttt{'5'}\ \$\ \mathit{Dva}\ 103\ 430 \\
&\qquad \rightsquigarrow \mathit{Dva}\ 105\ 450
\end{aligned}$$

(bez explicitního typu *asString* neví, že má vracet to samé *a*)

Prism

Prism je speciální případ *Traversal*, který matchne maximálně jednu hodnotu.

- *view* nemusí nevyžadovat monoidnost, stačí *preview* s *Maybe* (implementace pomocí funktoru *First*)
- jde obrátit (*review*, *re*)

Praktická definice pro implementaci: *Iso* totální jen v jednom směru.

Prism

preview *_Right* (*Right* 5) $\leadsto$ *Just* 5
preview *_Right* (*Left* 5) $\leadsto$ *Nothing*

view (*re* *_Right*) 3 $\leadsto$ *Right* 3
review *_Right* 3 $\leadsto$ *Right* 3

preview (*prefixed* `"tele"`) `"teleskop"` $\leadsto$ *Just* `"skop"`
preview (*prefixed* `"tele"`) `"pomeranc"` $\leadsto$ *Nothing*
review (*prefixed* `"tele"`) `"graf"` $\leadsto$ `"telegraf"`
over (*prefixed* `"tele"`) *reverse* `"telegraf"` $\leadsto$ `"telefarg"`
over (*prefixed* `"tele"`) *reverse* `"mrkev"` $\leadsto$ `"mrkev"`

Intuice: Prism generalizuje datový konstruktor a pattern match.

Lens library

Optika je implementovaná v *Control.Lens*.

Bonusy:

- TemplateHaskell
- Fajn operátory

Alternativní knihovny:

- *Lens.Micro* (méně funkcí, výrazně menší bloat)
- *Optics.Optic* (jednodušší encoding, mírnější typy, nespojuje se tečkou)
- *Prolens* (encoding v profunktorech, trochu ostřejší typy, lepší rychlost, zatím žádný bloat a prakticky žádné funkce)

@IMPUREPICS

HOW MANY LEVELS OF LENSES ARE YOU ON?

```
obj.getter()
obj.setters()
```

```
get: s -> a
set: a -> s -> s
```

```
type Lens s a =
∀f. Functor f
=> (a -> f a)
-> (s -> f s)
```

```
type Lens s t a b =
∀p. Strong p
=> p a b
-> p s t
```

Lens library — TH

TemplateHaskellem je možné automaticky vyrobit optiku pro ADT:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Dva a = Dva { _x :: a, _y :: a } deriving Show
makeLenses ''Dva
x :: Functor f ⇒ (a → f a) → Dva a → f (Dva a)
y :: Functor f ⇒ (a → f a) → Dva a → f (Dva a)
```

(podtržítka jsou konvence)

Lens library — TH

Kolize jmen se řeší pomocí typových tříd:

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}

import Control.Lens

data Dva a = Dva { _dvaX :: a, _dvaY :: a }
makeFields ''Dva

data Three a = Three { _threeX :: a, _threeY :: a, _threeZ :: a }
makeFields ''Three

x :: (Functor f, HasX s a) => (a -> f a) -> s -> f s
```

(podtržítka s názvem konstruktoru jsou konvence)

Následuje náhodná přehlídka užitečných lensových konstrukcí.

Operátory!

prefix	infix	State
view	`^.`	
set	`.~`	`.=`
over	`%~`	`%=`
toListOf	`^..`	
preview	`^?`	
	`+~, -~, *~`	`+=, -=, *=`

Pomůcka: `&` je obrácený `$`.

Operátory, pomůcky pro State

$$
\begin{aligned}
&(0,1,0,1,0,1)\ \&\ \_1\ .\tilde{}\ 7 \\
&\qquad\qquad \&\ \_2\ \%\tilde{}\ (5*) \\
&\qquad\qquad \&\ \_3\ .\tilde{}\ (-1) \\
&\qquad\qquad \&\ \_4\ .\tilde{}\ \texttt{"orange"} \\
&\qquad\qquad \&\ \_5\ +\tilde{}\ 2 \\
&\qquad\qquad \&\ \_6\ *\tilde{}\ 3 \\
&\quad \rightsquigarrow (7, 5, -1, \texttt{"orange"}, 2, 3)
\end{aligned}
$$

```
runPhysics :: State SomeWorld ()
runPhysics dT =
  zoom (gameObjects . actors) $
    do Vec x y <- use (player . speed)
       player . position . x += dT * x
       player . position . y += dT * y
```


MY INNOCENT
CODEBASE
@impurepics

import *Data.Aeson.Lens*

Prismy místo (ošklivých) patternmatchů:

```
"[1, \"x\"]" ^? nth 0 . _Number
```
⇝ *Just* 1.0

```
"[1, \"x\"]" ^? nth 1 . _Number
```
⇝ *Nothing*

```
"[10.5]" ^? nth 0 . _Integer
```
⇝ *Just* 10

```
"[{\"x\":1, \"y\":2}, {\"x\":3, \"y\":[]}]"
  ^.. values . members . _Integer
```
⇝ $[1, 2, 3]$

```
"[{\"x\":1, \"y\":2}, {\"x\":3, \"y\":4}]"
  ^.. values . key "x" . _Double
```
⇝ $[1.0, 3.0]$

Indexy, filtry, …

```
import Control.Lens.Indexed
import Control.Lens.Combinators
```

Indexované lensy obsahují ‘skrytý’ index který jde později použít.

```
['a'..'z'] ^.. itraversed . filtered (∈ "hello")
    ⇝ "ehlo"
['a'..'z'] ^.. itraversed . filtered (∈ "hello") . withIndex
    ⇝ [(4,'e'),(7,'h'),(11,'l'),(14,'o')]
['a'..'z'] ^.. itraversed . indices odd
    ⇝ "bdfhjlnprtvxz"
```

Fajn pomůcky:

```
has (element 10) [1,2,3]    ⇝ False
is _Left (Right 5)          ⇝ False
hasn't _Left (Right 5)      ⇝ True
```

```
"Some random words" & worded
  %~ show . length
    ⇝ "4 6 5"
"AAAAARGGG!" & _tail . mapped
  %~ toLower
    ⇝ "Aaaaarggg!"
```

Transformujeme monády

Motivace

Co když chceme monádu, která umí:

- State (*put*, *get*, ...) a IO (*print*)?
- Parsec + IO?
- IO + Either?
- State + IO + seznamový nedeterminismus? (a.k.a. prolog)

Software engineering: Naprogramujeme všechny kombinace!

Motivace

Co když chceme monádu, která umí:

- State (*put*, *get*, ...) a IO (*print*)?
- Parsec + IO?
- IO + Either?
- State + IO + seznamový nedeterminismus? (a.k.a. prolog)

Software engineering: Naprogramujeme všechny kombinace!

Při výrobě nové monády dostanu 2^n nových kombinací!

Lepší nápad

Nešly by existující monády kombinovat nějak jednoduše?

Příklad IO s možností selhat (*Maybe ∘ IO*):

```
newtype MaybeIO a =
  MaybeIO {runMaybeIO :: IO (Maybe a)}
instance Functor MaybeIO where
  fmap f (MaybeIO m) = MaybeIO $ fmap (fmap f) m
instance Applicative MaybeIO where
  pure = MaybeIO . pure . Just
  MaybeIO a <*> MaybeIO b = MaybeIO $
    a >>= maybe (return Nothing)
                ((<$>b) . fmap)
```

NB.: aplikativní instance je mírně asymetrická.

```
instance Monad MaybeIO where
  return = pure
  MaybeIO a >>= f = MaybeIO $
    a >>= maybe (return Nothing)
                (runMaybeIO . f)
```

MaybeIO

Jak použít MaybeIO?

$$runMaybeIO :: MaybeIO\ a \to IO\ (Maybe\ a)$$

Jak vyvolat IO akci:

$$liftIO :: IO\ a \to MaybeIO\ a$$
$$liftIO\ io = MaybeIO\ (return \langle\$\rangle\ io)$$

Jak vyvolat Maybe ‘akci’:

$$failMIO :: MaybeIO\ a$$
$$failMIO = MaybeIO\ (return\ Nothing)$$

Demo

```
io = liftIO
main = runMaybeIO $ do
  io $ putStrLn "daj cislo"
  a ← io (readLn :: IO Float)
  if a < 0 then failMIO
    else io $ putStrLn "jde odmocnit!"
  io . print $ sqrt a
```

Transformer

Pozorování: Nepoužili jsme nic specifického pro IO! Můžeme generalizovat pro všechny "obalené" monády.

```
newtype MaybeT m a =
  MaybeT {runMaybeT :: m (Maybe a)}

instance Functor m ⇒ Functor (MaybeT m) where
  fmap f = MaybeT · fmap (fmap f) · runMaybeT

instance Monad m ⇒ Applicative (MaybeT m) where
  pure = MaybeT · pure · Just
  MaybeT a ⟨∗⟩ MaybeT b = MaybeT $
    a ⪢= maybe (return Nothing)
               ((⟨$⟩b) · fmap)
```

Transformer

Pozorování: Nepoužili jsme nic specifického pro IO! Můžeme generalizovat pro všechny “obalené” monády.

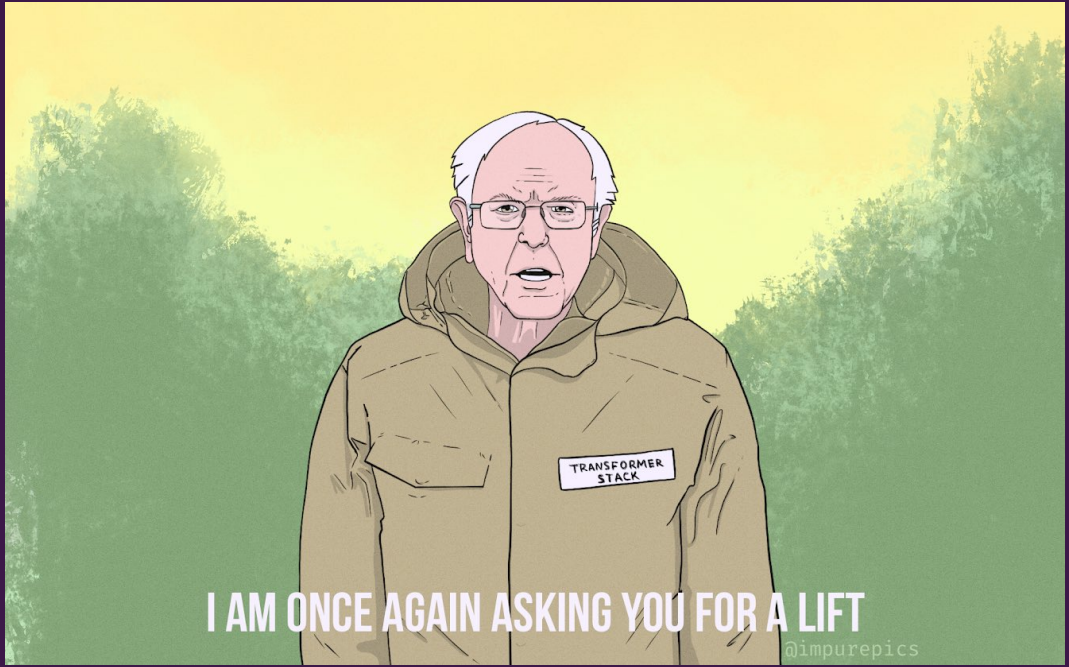
```
newtype MaybeT m a =
  MaybeT {runMaybeT :: m (Maybe a)}

instance Functor m ⇒ Functor (MaybeT m) where
  fmap f = MaybeT · fmap (fmap f) · runMaybeT

instance Monad m ⇒ Applicative (MaybeT m) where
  pure = MaybeT · pure · Just
  MaybeT a ⟨∗⟩ MaybeT b = MaybeT $
    a ⪢= maybe (return Nothing)
               ((⟨$⟩b) · fmap)
```

Transformer!

```
instance Monad m ⇒ Monad (MaybeT m) where
  return = pure
  MaybeT a >>= f = MaybeT $
    a >>= maybe (return Nothing)
                (runMaybeT · f)
```

Generický lift

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
instance MonadTrans MaybeT where
  lift = MaybeT · fmap Just
```


TRANSFORMER STACK
I AM ONCE AGAIN ASKING YOU FOR A LIFT
@impurepics

Příjemnější operace

Pamatovat si o kolik úrovní níž je v monádovém stacku zastrčené IO: nuda.

$$\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadIO}\ m \textbf{ where}\\
&\quad \mathit{liftIO} :: \mathit{IO}\ a \to m\ a\\
&\textbf{instance } \mathit{MonadIO}\ \mathit{IO} \textbf{ where}\\
&\quad \mathit{liftIO} = \mathit{id}\\
&\textbf{instance } (\mathit{MonadIO}\ m, \mathit{MonadTrans}\ t)\\
&\qquad\qquad \Rightarrow \mathit{MonadIO}\ (t\ m) \textbf{ where}\\
&\quad \mathit{liftIO} = \mathit{lift} \cdot \mathit{liftIO}
\end{aligned}$$

(tato implementace je naivní a ilustrativní)

Zjednodušení: $\mathit{putStrLn} :: \mathit{MonadIO}\ m \Rightarrow \mathit{String} \to m\ ()$

Ještě příjemnější operace

Samozřejmě existují *StateT*, *ReaderT*, *EitherT*, ...

Operace pro práci s monádou jsou typicky přetížené i pro transformované varianty, např v knihovně `mtl`:

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
  state :: (s → (a, s)) → m a
class (Monoid w, Monad m)
      ⇒ MonadWriter w m | m → w where
  tell :: w → m ()
class Monad m ⇒ MonadError e m | m → e where
  throwError :: e → m a
  catchError :: m a → (e → m a) → m a
```

Knihovní transformery a monády

Typické knihovny na monádové stacky jsou 2:

- `mtl` — starší, postavená z víceparametrových typových tříd
- `transformers` — novější, jednodušší, bez generických interfaců
- `mtl-tf` — generické interfacy nad `transformers` pomocí typových rodin

Doporučení (2021): `mtl` používejte jen z důvodů zpětné kompatibility

Běžné transformery

- *Identity* — nedělá nic, ale hodí se např. na odstraňování speciálních případů: *NejakaMonada* = *NejakaMonadaT Identity*
- *MaybeT*, *EitherT* (není vhodné na chyby!)
- *ExceptT* (*ErrorT* je deprecated)
- *StateT* — stavový výpočet
- *ReaderT* — výpočet s globálním prostředím, *ask*
- *WriterT* — vypočet konstruuje monoid, *tell*
- *RWST* — ReaderWriterStateT (poměrně typické kombo)
- *ListT* — funguje, ale má trochu nevhodný systém selhávání

Reader trivia

Instance $Monad\ (Reader\ r)$ **je prakticky ekvivalentní** $Monad\ ((\rightarrow)\ r)$**.**

O jakémkoliv rozšiřování parametrů se často mluví jako o Readeru.

Reader trivia

Instance $Monad\ (Reader\ r)$ **je prakticky ekvivalentní** $Monad\ ((\rightarrow)\ r)$**.**

O jakémkoliv rozšiřování parametrů se často mluví jako o Readeru.

Human-style: $\lambda x \rightarrow (f\ x, g\ x)$

Reader-style: $liftA2\ (,)\ f\ g$

Užitečné funktory

- *Identity*
- *Const c*
- *Compose f g* — 2 (aplikativní) funktory v jednom
- *Sum*, *Product*
- *Reverse* — obrací folding a traverzování
- *Backwards* — obrací pořadí operací u ($\langle * \rangle$)

Jak na backtracking?

Možnost 0: Alternative
(typicky pro parsery)

Možnost 1: MonadPlus

$$\textbf{class}\ (\mathit{Alternative}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}$$
$$\quad \mathit{mzero} :: m\ a$$
$$\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a$$

mplus se díky požadavku na monadičnost může chovat podstatně slušněji.

Jak na backtracking?

Možnost 2: *LogicT*

```
class MonadPlus m ⇒ MonadLogic m where
  msplit :: m a → m (Maybe (a, m a))    -- první výsledek
  ifte :: m a → (a → m b) → m b → m b    -- soft cut
  once :: m a → m a    -- hard cut
  (≫-) :: m a → (a → m b) → m b    -- férová konjunkce
  interleave :: m a → m a → m a    -- disjunkce
```

Zbytek:

- $(\gg=)$ je prologová čárka
- *guard* funguje jako pro seznamy
- '*mplus*' je prologový středník

CPS

Continuation passing style je metoda programování bez ‘vracení’ hodnot.

Normální kód: *f x* $=$ **do** $\{r \leftarrow g\ x; \mathit{return}\ (2 * r);\ \}$

CPS-style kód: *f x cont* $= g\ x\ (\lambda r \rightarrow \mathit{cont}\ (2 * r))$

Odpovídající *Cont* a *ContT* jdou použít (mimo jiné) k výraznému ovlivňování control-flow.

```
import Control.Monad
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

getCC = callCC (return . fix)

main = flip runContT return $ do
  lift $ putStrLn "nazdar!"
  restart ← getCC
  lift $ putStrLn "hadej cislo"
  a ← lift (readLn :: IO Int)
  unless (a ≡ 42) restart
  lift $ putStrLn "konecne dobre!"
```

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (MonadTrans\ t, Monad\ m) \Rightarrow m\ a \rightarrow t\ m\ a$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow SomeT\ xxx\ m\ a$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow SomeT\ xxx\ m\ a$$

$$liftSomeAction :: MonadSomeAction\ m \Rightarrow Some\ xxx\ a \rightarrow m\ a$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow SomeT\ xxx\ m\ a$$

$$liftIO :: MonadIO\ m \Rightarrow IO\ a \rightarrow m\ a$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow SomeT\ xxx\ m\ a$$

$$liftIO :: MonadIO\ m \Rightarrow IO\ a \rightarrow m\ a$$

$$\textbf{instance}\ (MonadSome\ m, MonadTrans\ t) \Rightarrow MonadSome\ (t\ m)$$

Shrnutí

$$runSomeT :: Monad\ m \Rightarrow SomeT\ xxx\ m\ a \rightarrow m\ (yyy\ a)$$

$$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow SomeT\ xxx\ m\ a$$

$$liftIO :: MonadIO\ m \Rightarrow IO\ a \rightarrow m\ a$$

$$\textbf{instance}\ (MonadIO\ m, MonadTrans\ t) \Rightarrow MonadIO\ (t\ m)$$

Extra příklad

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i − 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra příklad

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i − 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra příklad

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i − 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra příklad

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i − 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Nebezpečenství: Na pořadí transformerů záleží!

StateT s (ParserT IO) a

vs.

ParserT (StateT s IO) a

Nebezpečenství: Na pořadí transformerů záleží!

StateT s (ParserT IO) a

vs.

ParserT (StateT s IO) a

Důsledek: *IOT* **nemá úplně dobrý smysl.**

class *X a b c* | *a b* → *c* **where**

…

Funkcionální dependence:

- typy před šipkou jednoznačně určují typy za šipkou
- slouží k odstranění nejednoznačnosti

class *MonadState m s* | *m* → *s*
instance *Monad m* ⇒ *MonadState* (*StateT s m*) *s*

class *KeyValueContainer c k v* | *c* → *k v*
instance (*Map Int String*) *Int String*

class *Convertible a b*
instance *Convertible Int Float*

MPTCs vs. Prolog

Haskell:

```
data Z
data S a

class Number a
instance Number Z
instance Number a ⇒ Number (S a)

class Succ a b | a → b
instance Succ a (S a)

class Add a b c | a b → c
instance Add a b c ⇒ Add (S a) b (S c)
instance Add Z b b
```

Prolog:

```
number(z).
number(s(A)) :- number(A).

% succ(+A, -A1)
succ(X, s(X)).

% add(+X, +Y, -Z)
add(z, X, X).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

Dodatek: Type Families, associated types

```
class Monad m ⇒ MonadState m where
  type StateType m
  get :: m (StateType m)
  put :: StateType m → m ()
```

```
instance Monad m ⇒ MonadState (StateT s m) where
  type StateType (StateT s m) = s
  get = lift get
  put = lift · put
```

```
class KeyValueContainer a where
  type Key a
  type Value a
  keys :: a → [Key a]
  values :: a → [Value a]
```

```
instance KeyValueContainer [(a, b)] where
  type Key [(a, b)] = a
  type Value [(a, b)] = b
  keys = map fst
  values = map snd
```

Type Families vs. funkce

Typové rodiny jsou doslova funkce na typech.

```
ghci> :set -XTypeFamilies
ghci> data family X a
ghci> data instance X Int = String
ghci> data instance X Bool = Double
ghci> _ :: X Bool

<interactive>:13:1: error:
    - Found hole: _ :: X Bool
      ...
    - Valid hole fits include
        Double :: X Bool (defined at <interactive>:5:24)
```

Víceparametrové typové třídy a typové rodiny
jsou 2 různá vyjádření toho samého.

Bonus: V obojím jde velice jednoduše vytvořit nerozhodnutelný problém.

Typy jsou hodnoty

```
data Z
data S n

data family Succ a
data instance Succ a = S a
  -- ...

data Vec l where     -- GADT syntax
  Nil :: Vec Z
  Cons :: Int → Vec n → Vec (S n)
```

```
{-# LANGUAGE DataKinds #-}
data Nat = S Nat | Z

succ a = S a

data Vec :: Nat → ∗
  where
    Nil :: Vec Z
    Cons :: Int → Vec n → Vec (S n)
```

Výsledek: *vec3d* :: *Vec* (*S* (*S* (*S Z*)))

Pomocí *GHC.TypeLits* jde do typů dostat běžná čísla, stringy, apod.

Stringologie a formátování textu

Text v Haskellu

- String
  - [*Char*] — obrovský overhead na písmeno (16-32B), každé písmeno je boxed value, ...
  - každá položka seznamu je jeden nezávislý Unicode znak (tj. bez normalizace)
  - pro zpracování většího množství textu je String pomalý

Text v Haskellu

- String
  - [*Char*] — obrovský overhead na písmeno (16-32B), každé písmeno je boxed value, ...
  - každá položka seznamu je jeden nezávislý Unicode znak (tj. bez normalizace)
  - pro zpracování většího množství textu je String pomalý
- ByteString
  - obdoba `char*`, implementačně cca. *Vector Word8*
  - žádný Unicode!
  - vektorová rychlost, vstup Attoparsecu

Text v Haskellu

- String
  - [*Char*] — obrovský overhead na písmeno (16-32B), každé písmeno je boxed value, ...
  - každá položka seznamu je jeden nezávislý Unicode znak (tj. bez normalizace)
  - pro zpracování většího množství textu je String pomalý
- ByteString
  - obdoba `char*`, implementačně cca. *Vector Word8*
  - žádný Unicode!
  - vektorová rychlost, vstup Attoparsecu
- Text
  - Unicode-aware (interně uloženo jako normalizované UTF)
  - interně to je *Array*-slice
  - poměrně malý overhead všech Stringových operací

Text v Haskellu

- String
  - [*Char*] — obrovský overhead na písmeno (16-32B), každé písmeno je boxed value, ...
  - každá položka seznamu je jeden nezávislý Unicode znak (tj. bez normalizace)
  - pro zpracování většího množství textu je String pomalý
- ByteString
  - obdoba `char*`, implementačně cca. *Vector Word8*
  - žádný Unicode!
  - vektorová rychlost, vstup Attoparsecu
- Text
  - Unicode-aware (interně uloženo jako normalizované UTF)
  - interně to je *Array*-slice
  - poměrně malý overhead všech Stringových operací

IsString

Protože je těžké odhadnout, v jakém formátu chce programátor mít literály, existuje třída *IsString*:

```
{-# LANGUAGE OverloadedStrings #-}
class IsString a where
  fromString :: String → a

"ahoj" :: IsString s ⇒ s

"ahoj" :: Text       ⇝ OK
"ahoj" :: ByteString ⇝ OK
```

ByteString

Konverze:

import *qualified Data.ByteString as B*

$B.pack :: [Word8] \to ByteString$
$B.unpack :: ByteString \to [Word8]$

Postupná konstrukce:

$B.cons :: Word8 \to ByteString \to ByteString$
$B.snoc :: ByteString \to Word8 \to ByteString$
$B.uncons :: ByteString \to Maybe\ (Word8, ByteString)$
$B.unsnoc :: ByteString \to Maybe\ (ByteString, Word8)$

Interně: většina substringových operací na *ByteString*u je jen manipulace pointerů v O(1) nad jedním bufferem. Pokud chcete zahodit zbytek bufferu, použijte *B.copy*.

ByteString

I/O:

$$B.getLine :: IO\ ByteString$$
$$B.hGetLine :: Handle \to IO\ ByteString$$
$$B.hPutStr :: Handle \to ByteString \to IO\ ()$$
$$B.hGetContents :: Handle \to IO\ ByteString$$
$$B.hGet :: Handle \to Int \to IO\ ByteString$$
$$B.hPut :: Handle \to ByteString \to IO\ ()$$

apod.

UTF

Obsah *ByteString*u je pro převedení na String nebo Text potřeba *dekódovat*. (Unicode jde kódovat jako UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, …, ztrátově jako Latin1, ISO8859-1, Codepage 1250, KOI-8, …)

```
import Data.Text.Encoding

decodeUtf8 :: ByteString → Text
decodeUtf16BE :: ByteString → Text
decodeUtf32LE :: ByteString → Text

encodeUtf8 :: Text → ByteString
encodeUtf16LE :: Text → ByteString
```

Text

Text je hodně optimalizovaný (většina funkcí se fůzuje a/nebo zmizí).

```
import qualified Data.Text as T
T.pack :: String → Text
T.unpack :: Text → String
```

...

(*cons*, *snoc*, *append*, *take*, *takeEnd*, *intercalate*, *toLower*, *justifyLeft*, *splitOn*, ...)

IO-related funkce jsou v *Data.Text.IO*

Text

Text je hodně optimalizovaný (většina funkcí se fůzuje a/nebo zmizí).

```
import qualified Data.Text as T

T.pack :: String → Text
T.unpack :: Text → String
```

...

(*cons*, *snoc*, *append*, *take*, *takeEnd*, *intercalate*, *toLower*, *justifyLeft*, *splitOn*, ...)

IO-related funkce jsou v *Data.Text.IO*

Jak si vybrat?

situace	řešení
Potřebuju rychlost, polobinární obsah vstupu interpretuju vlastním kódem	*ByteString*
Používám AttoParsec	*ByteString*
Mám nekonečnou větu plnou nerozhodnutelných písmen	*String*
Aplikace je jednoduchá a stringový overhead je skoro nulový	*String*
Všechny ostatní případy	*Text*

Pravda o Pretty-Printingu

Pravda o Pretty-Printingu

Show **není pretty-printing!**

Pretty-printing se od *Show*-ování liší v několika podstatných ohledech:

- výstup není potřeba parsovat (nemusí obsahovat detaily)
- výstup může obsahovat redundanci
- občas je potřeba odsazovat a zarovnávat

Text.PrettyPrint

Pretty-printing se od *Show*-ování liší v několika podstatných ohledech:

- výstup není potřeba parsovat (nemusí obsahovat detaily)
- výstup může obsahovat redundanci
- občas je potřeba odsazovat a zarovnávat

Haskellový přístup:

```
import Text.PrettyPrint
renderStyle (style {lineLength = 80}) :: Doc → String
```

Text.PrettyPrint

Pretty-printing se od *Show*-ování liší v několika podstatných ohledech:

- výstup není potřeba parsovat (nemusí obsahovat detaily)
- výstup může obsahovat redundanci
- občas je potřeba odsazovat a zarovnávat

Haskellový přístup:

```
import Text.PrettyPrint
renderStyle (style {lineLength = 80}) :: Doc → String
```

Doc je abstrakce pro různě pospojovaná políčka textu (podobně jako TEX-ové vlisty/hlisty).

$$
\begin{aligned}
&\mathit{empty} :: \mathit{Doc} \\
&\mathit{text} :: \mathit{String} \to \mathit{Doc} \\
&\mathit{sizedText} :: \mathit{Int} \to \mathit{String} \to \mathit{Doc} \\
&\mathit{int} :: \mathit{Int} \to \mathit{Doc} \\
&\mathit{float} :: \mathit{Float} \to \mathit{Doc}
\end{aligned}
$$

Dekorace:

$$semi, equals, lparen, rparen, comma, colon, space, \ldots :: Doc$$

$$parens, brackets, quotes, \ldots :: Doc \to Doc$$

$$punctuate :: Doc \to [Doc] \to [Doc]$$

Lepení:

$$
\begin{array}{lll}
(\diamond) & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- vedle} \\
(\langle + \rangle) & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- vedle s mezerou (nějakou)} \\
(\$\$) & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- nad sebou} \\
(\$+\$) & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- nad sebou bez nestingu} \\
\mathit{nest} & :: \mathit{Int} \to \mathit{Doc} \to \mathit{Doc} & \text{-- odsazení}
\end{array}
$$

Seznamové verze: *hcat*, *hsep*, *vcat*, *vsep*

Chytré verze (v/h): *cat*, *sep*

Odstavcové verze: *fcat*, *fsep*

Text.PrettyPrint — demo

```
import Text.PrettyPrint

data Scheme = Ident String | Number Int | Seq [Scheme]

class PDoc a where
  pdoc :: a → Doc

ppshow :: PDoc a ⇒ a → String
ppshow = renderStyle (style {lineLength = 80}) · pdoc

instance PDoc Scheme where
  pdoc (Ident a) = text a
  pdoc (Number i) = int i
  pdoc (Seq xs) = parens $ sep (map pdoc xs)
```

Text.PrettyPrint — demo

$$short\ k = Seq\ \$\ Ident\ \texttt{"*"} : map\ Number\ [k\,..\,k+3]$$
$$long = Seq\ \$\ Ident\ \texttt{"+"} : map\ short\ [1\,..\,10]$$
$$main = putStrLn \cdot ppshow\ \$\ Seq\ [Ident\ \texttt{"factorial"}, long]$$

```
(factorial
 (+
  (* 1 2 3 4)
  (* 2 3 4 5)
  (* 3 4 5 6)
  (* 4 5 6 7)
  (* 5 6 7 8)
  (* 6 7 8 9)
  (* 7 8 9 10)
  (* 8 9 10 11)
  (* 9 10 11 12)
  (* 10 11 12 13)))
```

$$pdoc\ (Seq\ [\,]) = parens\ empty$$
$$pdoc\ (Seq\ (x : xs)) = parens\ \$\ pdoc\ x\ \langle + \rangle\ sep\ (map\ pdoc\ xs)$$

```
(factorial (+ (* 1 2 3 4)
              (* 2 3 4 5)
              (* 3 4 5 6)
              (* 4 5 6 7)
              (* 5 6 7 8)
              (* 6 7 8 9)
              (* 7 8 9 10)
              (* 8 9 10 11)
              (* 9 10 11 12)
              (* 10 11 12 13)))
```

IO!

Co ještě umí run-time Haskellu?

- UNIX-like soubory a síťovou komunikaci
- konkurentní výpočet programu a synchronizaci
- paralelní výpočet programu
- reference a pointery
- FFI

Komunikace uživatele s programem

- Proměnlivé prostředí (‘env’)

```
import System.Environment

getEnv :: String → IO String
lookupEnv "PATH" :: IO (Maybe String)
getEnvironment :: IO [(String, String)]
```

Komunikace uživatele s programem

- Proměnlivé prostředí (‘env’)

```
import System.Environment

getEnv :: String → IO String
lookupEnv "PATH" :: IO (Maybe String)
getEnvironment :: IO [(String, String)]
```

- Parametry z příkazového řádku

```
getArgs :: IO [String]
getProgName :: IO String
getExecutablePath :: IO FilePath
```

Komunikace uživatele s programem

- exit-code

```
import System.Exit

die "sorry jako" :: IO a

exitWith (exitSuccess) :: IO a
exitWith (exitFailure 3) :: IO a
```

Komunikace uživatele s programem

- exit-code

```
import System.Exit

die "sorry jako" :: IO a

exitWith (exitSuccess) :: IO a
exitWith (exitFailure 3) :: IO a
```

- čtení a zápis z/do souborových deskriptorů

```
import System.IO

openFile "test" WriteMode :: IO Handle
hClose h :: IO ()

hGetChar h :: IO Char
hPutChar h 'c' :: IO ()
```

Komunikace uživatele s programem

- posílat a chytat signály

```
import System.Posix.Signals

raiseSignal sigSTOP :: IO ()
signalProcess sigKILL 1 :: IO ()
```

Komunikace uživatele s programem

- posílat a chytat signály

```
import System.Posix.Signals

raiseSignal sigSTOP :: IO ()
signalProcess sigKILL 1 :: IO ()


data Handler = Default | Ignore
    | Catch (IO ()) | CatchOnce (IO ()) | ...
installHandler sigTERM
   (Catch $ putStrLn "nope")
   Nothing
    :: IO Handler
```

(většina ostatních nízkoúrovňových věcí je v balíčku `unix`)

Jak se pozná kvalitní UNIXový program?

Jak se pozná kvalitní UNIXový program?

Vypisuje `--help` související s realitou.

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

```
import Options.Applicative
import Data.Semigroup ((<>))

theOption :: Parser Bool
theOption = switch $ long "good"
                  <> short 'g'
                  <> help "Should it be really good?"

optInfo = info (helper <*> theOption)
  $ progDesc "good shows current level of goodness"
  <> header "A program that shows if it's good."
  <> footer "See more information on www.good.program.hu"
```

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

```
import Options.Applicative
import Data.Semigroup ((<>))

theOption :: Parser Bool
theOption = switch $ long "good"
                  <> short 'g'
                  <> help "Should it be really good?"

optInfo = info (helper <*> theOption)
  $ progDesc "good shows current level of goodness"
  <> header "A program that shows if it's good."
  <> footer "See more information on www.good.program.hu"
```

Abstrakce: Volitelné vlastnosti parametrů budeme popisovat jako monoid. Parametry pak aplikativně spojíme do velké struktury.

```
import Options.Applicative
import Data.Semigroup ((<>))

theOption :: Parser Bool
theOption = switch $ long "good"
                  <> short 'g'
                  <> help "Should it be really good?"

optInfo = info (helper <*> theOption)
  $ progDesc "good shows current level of goodness"
  <> header "A program that shows if it's good."
  <> footer "See more information on www.good.program.hu"
```

```
main = do opt ← execParser $ optInfo
          putStrLn $ if opt
                     then "yeah it's good"
                     else "not good at all!"
```

```
$ ./good
not good at all!
$ ./good -x
Invalid option '-x'

Usage: ./good [-g|--good]
$ ./good -g
yeah it's good
```

```
main = do opt ← execParser $ optInfo
          putStrLn $ if opt
                     then "yeah it's good"
                     else "not good at all!"
```

```
$ ./good
not good at all!
$ ./good -x
Invalid option '-x'

Usage: ./good [-g|--good]
$ ./good -g
yeah it's good
```

```
$ ./good -h
A program that shows if it's good.

Usage: good [-g|--good]
  good shows current level of goodness

Available options:
  -h,--help                Show this help text
  -g,--good                Should it be really good?

See more information on www.good.program.hu
```

- Stringy

```
optOutput = strOption
  $ long "output"
  ◇ short 'o'
  ◇ metavar "FILE"
  ◇ value "out.txt"
  ◇ showDefault
  ◇ help "Write output to FILE"
```

- Integery

```
optLevel = option
  $ long "level"
  ◇ short 'l'
  ◇ help "Level of whatever"
```

- Kombinace (produkt)

$$\begin{aligned}&\textbf{data}\ \mathit{CmdLineOpts} = \mathit{CmdLineOpts}\ \{\mathit{level} :: \mathit{Int},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{output} :: \mathit{String}\}\\ &\mathit{allOptions} = \mathit{CmdLineOpts}\ \langle\$\rangle\ \mathit{optLevel}\ \langle*\rangle\ \mathit{optOutput}\end{aligned}$$

- Kombinace alternativ

$$\begin{aligned}&\textbf{data}\ \mathit{OptionalOutput} = \mathit{FileOutput}\ \mathit{String} \mid \mathit{StdOutput}\\ &\mathit{optionOptionalOutput} = \mathit{FileOutput}\ \langle\$\rangle\ \mathit{optOutput}\\ &\quad\langle|\rangle\ \mathit{flag'}\ \mathit{StdOutput}\ (\mathit{long}\ \texttt{"stdout"}\\ &\qquad\qquad\qquad\qquad\diamond\ \mathit{help}\ \texttt{"write to stdout"})\end{aligned}$$

RTS

Run-time systém nad nízkoúrovňovým systémovým interfacem provozuje vlastní event loop.

- Umožňuje velmi efektivní konkurentní programování (“green threads”)
- Zabraňuje potížím s pollováním bez overheadu vláken
- Poskytuje C-like sdílení zapisovatelných objektů

ForkIO

```
import Control.Concurrent
forkIO :: IO () → IO ThreadID
```

forkIO provede IO akci paralelně s volající monádou.

Výhoda: Lehká vlákna nemají žádný overhead.

Pro komunikaci mezi vlákny máme (mimo jiné) *MVar*:

```
import Control.Concurrent.MVar

newEmptyMVar :: IO (MVar a)
newMVar :: a → IO (MVar a)

takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
readMVar :: MVar a → IO a
swapMVar :: MVar a → a → IO a
```

MVar může být plná nebo prázdná, put/take jsou blokující.

Konkurentní programování — příklad

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.MVar

main = do
  kolik ← readLn :: IO Int
  com ← newEmptyMVar
  forkIO $ do
    forM_ [1 .. kolik] $ λi → putMVar com i              ≫ threadDelay 333333
    forM_ [1 .. kolik] $ λi → putMVar com (kolik − i) ≫ threadDelay 333333
  let loop = do a ← takeMVar com
                when (a > 0) $ print a ≫ loop
  loop
```

Konkurentní programování — TCP server

```
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

Konkurentní programování — TCP server

```
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

Konkurentní programování — TCP server

```
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

Konkurentní programování — TCP server

```
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

```
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

```
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

Konkurentní programování — TCP server

```
serverLoop counter sock =
  forever $ do
    (conn, peer) ← accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (λ_ → close conn)
```

```
serverLoop counter sock =
  forever $ do
    (conn, peer) ← accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (λ_ → close conn)
```

```
serverLoop counter sock =
  forever $ do
    (conn, peer) ← accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (λ_ → close conn)
```

```
serverLoop counter sock =
  forever $ do
    (conn, peer) ← accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (λ_ → close conn)
```

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    -- String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET": _ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    -- String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET": _ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    -- String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET": _ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

Konkurentní programování — TCP server

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    -- String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET": _ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

IO Bonusy

Overhead *MVar* je moc velký/zbytečný

- *TVar* (neprázdná proměnná, atomické operace nad *STM*)
- *IORef* (neprázdná proměnná)
- *Ptr* (ošklivá Cčková reference na P.O.D. bez GC)
- *STM* (Software Transactional Memory monad) — poskytuje serializovaný přístup k paměti sdílené mezi vlákny *bez IO*.
- *Chan*, *TChan*, *QSem*, *QSemN*, ...

FFI

Když je potřeba sáhnout na dno, Haskell podporuje přímé napojení na systémový C-style kód pomocí FFI.

Typické použití:

- Obalování systémových API
- Obalování Cčkových knihoven (‘bindings’)
- Drcení čísel

FFI

FFI má 2 módy použití:

import
- Cčkový `.h` a `.c` soubor se automaticky skompilují a slinkují s programem
- funkce lze popsat jako *IO* akce, i jako čisté funkce
- typy parametrů se mapují z primitivních haskellových typů

export
- export haskellové funkce vygeneruje `.h`
- odpovídající symbol je ve skompilovaném `.o`

FFI — import

Haskell:

```
{-# INCLUDE "rawread.h" #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.C
foreign import ccall "raw_read" rawRead :: Word → IO Word
main = rawRead 0 >>= print
```

`rawread.h`: `int64_t raw_read(int64_t);`

`rawread.c`: (BONUS: skoro validní použití reinterpret_cast)

```
int64_t raw_read(int64_t addr) {
  int64_t* ptr=(int64_t*)addr;
  return *ptr;
}
```

FFI — import

Haskell:

```
{-# INCLUDE "rawread.h" #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.C
foreign import ccall "raw_read" rawRead :: Word → IO Word
main = rawRead 0 >>= print
```

`rawread.h`: `int64_t raw_read(int64_t);`

`rawread.c`: (BONUS: skoro validní použití reinterpret_cast)

```
int64_t raw_read(int64_t addr) {
  int64_t* ptr=(int64_t*)addr;
  return *ptr;
}
```

FFI — export

`Factorial.hs`:

```
import Foreign.C
foreign export ccall "fact" factorial :: Int → Int
factorial n = product [1..n]
```

`factorial.c`:

```
#include "Factorial_stub.h"
void someF() {
  printf("%d\n", fact(5));
}
```

Paralelní vyhodnocování

Paralelizace Haskellu je (technicky) celkem jednoduchá

- Většina jevů je bez efektů
- RTS umí vlastní scheduling ‘zadarmo’
- GC je celkem moderní a víc vláken zvládá dobře

  (narozdíl od Javy, C# a Pythonu)

Implementace:

Paralelní vyhodnocování

Paralelizace Haskellu je (technicky) celkem jednoduchá

- Většina jevů je bez efektů
- RTS umí vlastní scheduling ‘zadarmo’
- GC je celkem moderní a víc vláken zvládá dobře

(narozdíl od Javy, C# a Pythonu)

Implementace:

$$par :: a \to b \to b$$

“Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

a ‘par‘ b is exactly equivalent semantically to b.”

$$
\begin{aligned}
&\mathit{hardFunction} = \textbf{let} \\
&\quad \mathit{firstHalf} = \ldots \\
&\quad \mathit{secondHalf} = \ldots \\
&\quad \textbf{in } \mathit{firstHalf}\ `\mathit{par}`\ \mathit{secondHalf}\ `\mathit{par}`\ \mathit{firstHalf} + \mathit{secondHalf}
\end{aligned}
$$

Kompilace: `ghc -threaded -rtsopts hard.hs`

Spuštění pro 16 jader: `./hard +RTS -N16`

Nevýhoda:

$$
\begin{aligned}
&\mathit{hardFunction} = \textbf{let} \\
&\quad \mathit{firstHalf} = \ldots \\
&\quad \mathit{secondHalf} = \ldots \\
&\quad \textbf{in } \mathit{firstHalf}\ \text{`}\mathit{par}\text{`}\ \mathit{secondHalf}\ \text{`}\mathit{par}\text{`}\ \mathit{firstHalf} + \mathit{secondHalf}
\end{aligned}
$$

Kompilace: `ghc -threaded -rtsopts hard.hs`

Spuštění pro 16 jader: `./hard +RTS -N16`

Nevýhoda: Pokud paralelismem chceme dosáhnout rychlosti, je lepší program napsat v C a volat přes FFI. (paralelně)

Lepší kontrola nad prováděním kódu:

```
import Control.Parallel.Strategies
```

Lepší kontrola nad prováděním kódu:

```
import Control.Parallel.Strategies
```

Akcelerace automatickým přeložením programu do paralelního SIMD nebo CUDA:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Lepší kontrola nad prováděním kódu:

```
import Control.Parallel.Strategies
```

Akcelerace automatickým přeložením programu do paralelního SIMD nebo CUDA:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Lepší kontrola nad prováděním kódu:

```
import Control.Parallel.Strategies
```

Akcelerace automatickým přeložením programu do paralelního SIMD nebo CUDA:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Numerické výpočty s možností paralelizace:

```
import Data.Array.Repa
```

Lepší kontrola nad prováděním kódu:

```
import Control.Parallel.Strategies
```

Akcelerace automatickým přeložením programu do paralelního SIMD nebo CUDA:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Numerické výpočty s možností paralelizace:

```
import Data.Array.Repa
```

✎Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming (Simon Marlow, 2013)

Grafické knihovny

Jak na grafický výstup?

Samozřejmě existují bindingy na všechny možné GUI knihovny.

- gtk, Qt, wxWidgets
- SDL, OpenGL

Haskellově specifické věci jsou zajímavější:

- Back-end: JuicyPixels (&repa)
- Generativní grafika: Rasterific
- Specifická generativní grafika: Chart
- Interaktivní generativní grafika: Gloss & NotGloss

```
import Codec.Picture
readImage :: FilePath → IO (Either String DynamicImage)
```

DynamicImage je union pro všechny možné reprezentace pixelů, např. *ImageRGBA8* (*Image PixelRGBA8*) nebo *ImageYCbCr8* (*Image PixelYCbCr8*).

```
writeDynamicBitmap ::
  FilePath → DynamicImage → IO (Either String Bool)
writeBitmap :: BmpEncodable pixel ⇒
  FilePath → Image pixel → IO ()
writePng :: PngSavable pixel ⇒
  FilePath → Image pixel → IO ()

  ⋮
```

Co s obrázkem?

Vyrobit obrázek z funkce:

$$generateImage :: Pixel\ px \Rightarrow (Int \to Int \to px) \to Int \to Int \to Image\ px$$

map-like zjednodušení:

$$pixelMap :: Pixel\ a, Pixel\ b \Rightarrow (a \to b) \to Image\ a \to Image\ b$$

Pixely

Pixely je možné manipulovat přímo jako číselné hodnoty.

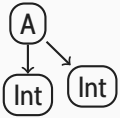
```
data PixelRGB8 = PixelRGB8 ! Pixel8 ! Pixel8 ! Pixel8
type PixelF = Float
data PixelCMYK16 = PixelCMYK16 ! Pixel16 ! Pixel16
                               ! Pixel16 ! Pixel16
```

Strict&Unboxed

Vykřičník v datovém typu znamená, že data nejsou ‘boxeď. (Tj. nemůžou být méně definovaná než celá struktura.)

Praktický význam:

data $A = A\ Int\ Int$



data $A = A\ !\ Int\ !\ Int$

Pointery navíc jsou klíčový rozdíl pro rychlost výpočtů na velkém množství malých dat.

Jak tomu pomoct?

$$seq :: a \to b \to b$$

“The value of seq a b is bottom if a is bottom, and otherwise equal to b. In other words, it evaluates the first argument a to weak head normal form (WHNF). seq is usually introduced to improve performance by avoiding unneeded laziness.”

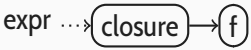
$f\ !\ a = \ldots$

$f\ \$!\ a$

```
{-# INLINE f #-}
```

Jak to vypadá v paměti?

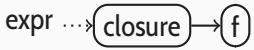
$$f :: X \to A$$
$$expr = f$$



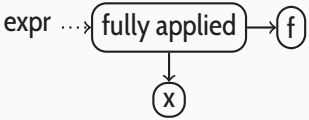
Jak to vypadá v paměti?

$f :: X \to A$

$expr = f$



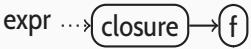
$expr = f\ x$



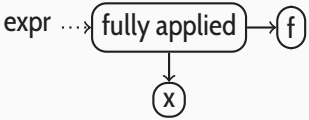
Jak to vypadá v paměti?

$f :: X \to A$

$expr = f$

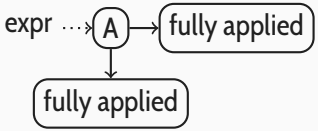


$expr = f\ x$



data $A = A\ Int\ Int$

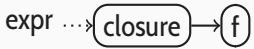
$expr = f\ \$!\ x$



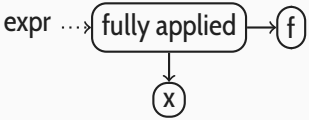
Jak to vypadá v paměti?

$f :: X \to A$

$expr = f$

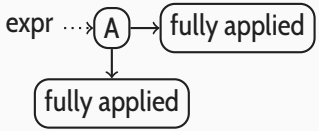


$expr = f\ x$



data $A = A\ Int\ Int$

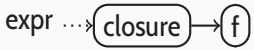
$expr = f\ \$!\ x$



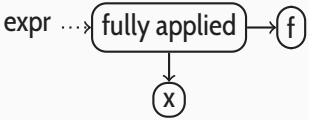
(mezikrok: *deepSeq*)

Jak to vypadá v paměti?

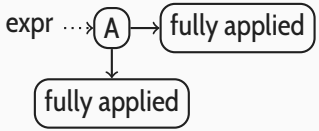
$f :: X \to A$
$expr = f$



$expr = f\ x$



data $A = A\ Int\ Int$
$expr = f\ \$!\ x$



(mezikrok: *deepSeq*)

data $A = A\ !\ Int\ !\ Int$
$expr = f\ \$!\ x$

Strict&Unboxed (prakticky)

Psát všude vykřičníky je otrava. Většina používaných kontejnerů má striktní varianty:

- *Data.Vector.Unboxed*, *Data.Vector.Primitive*
- *Data.Set.Strict*, *Data.Map.Strict*, …
- *Control.Monad.Trans*:
  - *State.Strict*, *Writer.Strict*, …
  - *RWS.Strict*
- *strict* :: *Strict lazy strict* $\Rightarrow$ *Iso′ lazy strict*
- *unpackStrict* :: *ByteString* $\rightarrow$ [*Word8*]
- *foldl′*

Repa

‘Regular parallel arrays’ (polymorfní vícedimenzionální primitivní striktní arraye)

- regularita $\rightarrow$ rychlost
- fůzování arrayových funkcí $\rightarrow$ víc rychlosti
- lowlevelové optimalizace při překladu s LLVM $\rightarrow$ ještě víc rychlosti
- paralelní vyhodnocování skoro zadarmo

type *DIM0* $=$ *Z*
type *DIM1* $=$ *DIM0* : . *Int*
type *DIM2* $=$ *DIM1* : . *Int*
...

- *R.Array U DIM2 Float* — matice
- *R.Array V Z Int* — 1 bod
- $(Z : . 1 : . 2 : . 3)$ — index ve 3D arrayi
- $(Z : . 3)$ — index v 1D arrayi

```
map :: (Shape sh, Source r a) ⇒
  (a → b) → Array r sh a
   → Array D sh b
zipWith :: (Shape sh, Source r1 a, Source r2 b) ⇒
  (a → b → c) → Array r1 sh a → Array r2 sh b
   → Array D sh c
fromFunction ::
  sh → (sh → a) → Array D sh a
```

Výpočet:

```
computeUnboxedS :: (Load r1 sh e, Unbox e) ⇒
  Array r1 sh e → Array U sh e
computeUnboxedP :: (Load r1 sh e, Monad m, Unbox e) ⇒
  Array r1 sh e → m (Array U sh e)
```

$$
\begin{aligned}
&\mathit{map} :: (\mathit{Shape}\ \mathit{sh}, \mathit{Source}\ r\ a) \Rightarrow \\
&\quad (a \to b) \to \mathit{Array}\ r\ \mathit{sh}\ a \\
&\quad \to \mathit{Array}\ D\ \mathit{sh}\ b \\
&\mathit{zipWith} :: (\mathit{Shape}\ \mathit{sh}, \mathit{Source}\ r1\ a, \mathit{Source}\ r2\ b) \Rightarrow \\
&\quad (a \to b \to c) \to \mathit{Array}\ r1\ \mathit{sh}\ a \to \mathit{Array}\ r2\ \mathit{sh}\ b \\
&\quad \to \mathit{Array}\ D\ \mathit{sh}\ c \\
&\mathit{fromFunction} :: \\
&\quad \mathit{sh} \to (\mathit{sh} \to a) \to \mathit{Array}\ D\ \mathit{sh}\ a
\end{aligned}
$$

Výpočet:

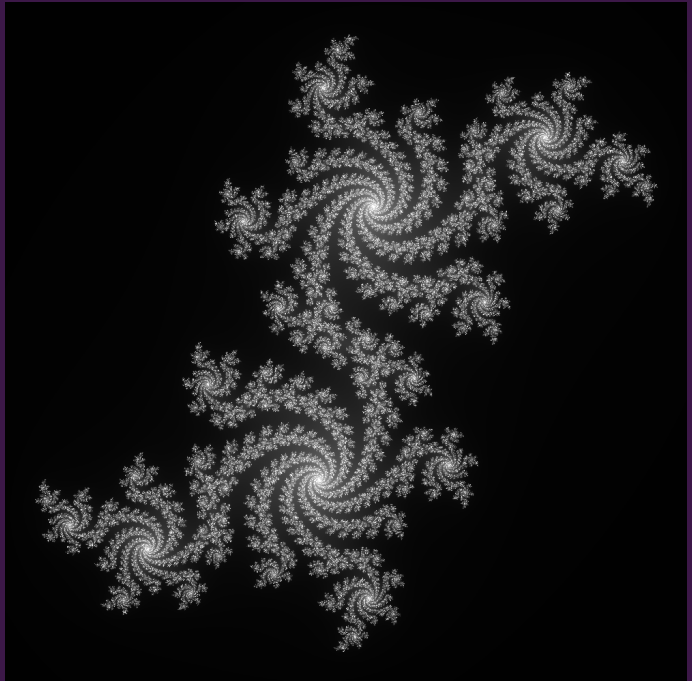
$$
\begin{aligned}
&\mathit{computeUnboxedS} :: (\mathit{Load}\ r1\ \mathit{sh}\ e, \mathit{Unbox}\ e) \Rightarrow \\
&\quad \mathit{Array}\ r1\ \mathit{sh}\ e \to \mathit{Array}\ U\ \mathit{sh}\ e \\
&\mathit{computeUnboxedP} :: (\mathit{Load}\ r1\ \mathit{sh}\ e, \mathit{Monad}\ m, \mathit{Unbox}\ e) \Rightarrow \\
&\quad \mathit{Array}\ r1\ \mathit{sh}\ e \to m\ (\mathit{Array}\ U\ \mathit{sh}\ e)
\end{aligned}
$$

Repa (použití)

```
ghc -O -fllvm -threaded -rtsopts program.hs
./program +RTS -N 16
```

Příklad (zpět k JuicyPixels)

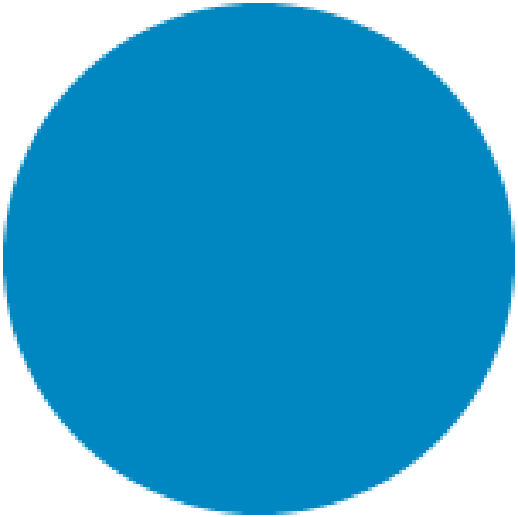
```
import Data.Complex
import Data.Word
import Codec.Picture

julia limit z a0 = run a0 0
  where run c i | i >= limit = limit
                | magnitude c > 2 = i
                | otherwise = run (c * c + z) (i + 1)

juliapix x y = (fromIntegral :: Int -> Word8) $
  julia 255 (0.141 :+ 0.6) $
  ((-1.25) :+ (-1.25)) + 0.0025 * (fromIntegral x :+ fromIntegral y)

main = writePng "fractal.png" $ generateImage juliapix 1000 1000
```



Jak nakreslit něco složitějšího?

```
import Codec.Picture (PixelRGBA8 (..), writePng)
import Graphics.Rasterific

white = PixelRGBA8 255 255 255 255
drawColor = PixelRGBA8 0 0x86 0xc1 255

main = writePng "kulaty.png" $
  renderDrawing 200 200 white $
    fill $ circle (V2 100 100) 75
```

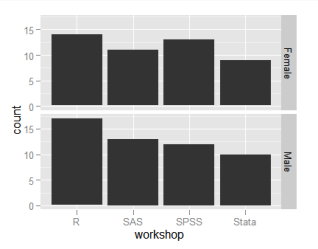
Další knihovny: `diagrams` (vektorové diagramy, zarovnávání, šipky, ...), `juicypixels-repa` (brutálnější výpočty), `friday` (běžné rastrové výpočty typu blur)

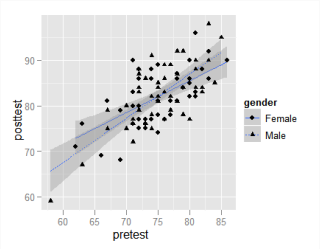


Grafy

Generovat pěkné grafy je tradičně těžké.

State-of-art: `ggplot2`





GGplot:

```
plot(data) + geom_point(pretest,posttest) + geom_quantile() + title(...)
```

Haskell: Jednotlivé kusy grafu jsou substruktury, můžeme do nich vrtat pomocí čoček.

Chart

```
import Graphics.Rendering.Chart
import Data.Colour
import Data.Colour.Names
import Data.Default.Class
import Graphics.Rendering.Chart.Backend.Cairo
import Control.Lens

setLinesBlue :: PlotLines a b → PlotLines a b
setLinesBlue = plot_lines_style . line_color .~ opaque blue

chart = toRenderable layout
  where
    am :: Double → Double
    am x = (sin (x * 3.14159 / 45) + 1) / 2 * (sin (x * 3.14159 / 5))

    sinusoid1 = plot_lines_values .~ [[(x, (am x)) | x ← [0, (0.5) .. 400]]]
              $ plot_lines_style . line_color .~ opaque blue
              $ plot_lines_title .~ "am" $ def

    sinusoid2 = plot_points_style .~ filledCircles 2 (opaque red)
              $ plot_points_values .~ [(x, (am x)) | x ← [0, 7 .. 400]]
              $ plot_points_title .~ "am points" $ def

    layout = layout_title .~ "Amplitude Modulation"
           $ layout_plots .~ [toPlot sinusoid1, toPlot sinusoid2] $ def

main = renderableToFile def "example1_big.png" chart
```

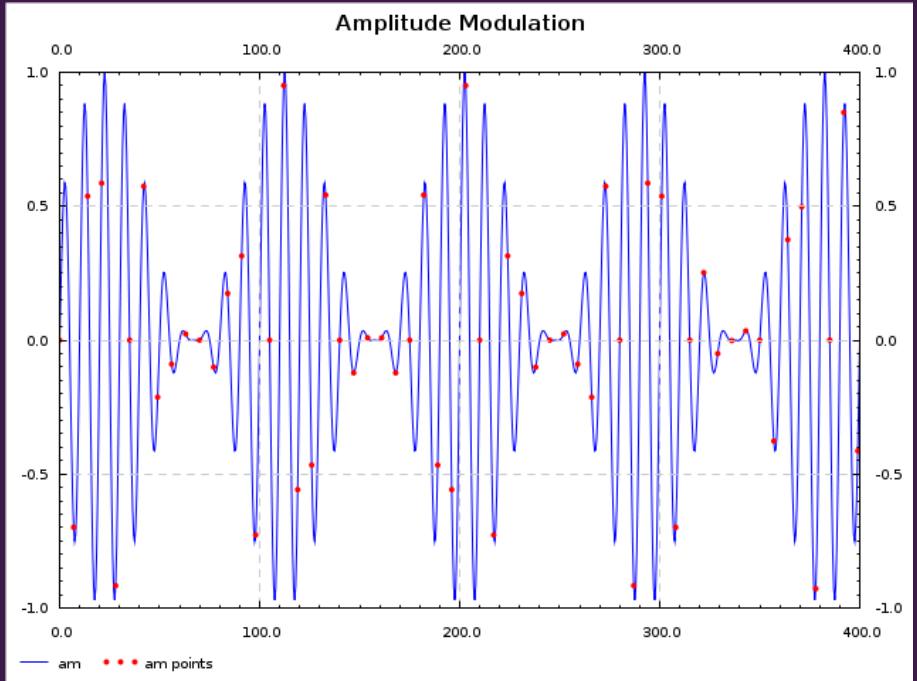

Amplitude Modulation
0.0
100.0
200.0
300.0
400.0
1.0
0.5
0.0
-0.5
-1.0
am
am points

Chart (Monads!)

```
import Graphics.Rendering.Chart.Easy
import Graphics.Rendering.Chart.Backend.Cairo

signal :: [Double] -> [(Double, Double)]
signal xs = [(x, (sin (x * 3.14159 / 45) + 1)
                 / 2 * (sin (x * 3.14159 / 5)))
            | x <- xs]

main = toFile def "example1_big.png" $ do
  layout_title .= "Amplitude Modulation"
  setColors [opaque blue, opaque red]
  plot (line "am" [signal [0,(0.5)..400]])
  plot (points "am points" (signal [0,7..400]))
```

“Get something cool on the screen in under 10 minutes.”

Gloss

Gloss poskytuje jednoduchý datový typ na popis scény a poměrně efektivní zobrazování scény pomocí OpenGL.

Scéna:

```
type Point = (Float, Float)
type Path = [Point]
data Picture =
    | Blank
    | Polygon Path | Line Path | Circle Float
    | Text String
    | Color Color Picture
    | Translate Float Float Picture
    | Rotate Float Picture | Scale Float Picture
    | ...
```

Je to rychlé?

Je to rychlé?

Je to lazy.

Gloss

```
import Graphics.Gloss
display :: Display → Color → Picture → IO ()
main = display (InWindow "Okno!" (200, 200) (10, 10))
  white (ThickCircle 50 3)

animate :: Display → Color → (Float → Picture) → IO ()
main = animate FullScreen
  white $ λt → Pictures $ flip map [1 .. 4] $ λx →
  Rotate (90 * x + 20 * t) $
  Translate 80 0 $
  ThickCircle 20 (3 + 2 * sin t)
```

```
simulate :: Display
         → Color
         → Int          -- fps
         → model        -- cokoliv
         → (model → Picture)
         → (ViewPort → Float → model → model)
         → IO ()
```

```
play :: Display
     → Color
     → Int    -- fps
     → world    -- popis světa
     → (world → Picture)    -- render
     → (Event → world → world)
     → (Float → world → world)
     → IO ()
```

3D: not-gloss

```
data VisObject a = VisObjects [VisObject a]
    | Trans (V3 a) (VisObject a)
    | RotQuat (Quaternion a) (VisObject a)
    | RotEulerDeg (Euler a) (VisObject a)
    | Scale (a, a, a) (VisObject a)
    | Cylinder (a, a) GlossColor.Color
    | Box (a, a, a) Flavour GlossColor.Color
    | Ellipsoid (a, a, a) Flavour GlossColor.Color
    | Line (Maybe a) [V3 a] GlossColor.Color
    | Arrow (a, a) (V3 a) GlossColor.Color
    | Plane (V3 a) GlossColor.Color GlossColor.Color
    | Triangle (V3 a) (V3 a) (V3 a) GlossColor.Color
    | Quad (V3 a) (V3 a) (V3 a) (V3 a) GlossColor.Color
    | Text3d String (V3 a) BitmapFont GlossColor.Color
    | Text2d String (a, a) BitmapFont GlossColor.Color
    | Points [V3 a] (Maybe GLfloat) GlossColor.Color
    | ObjModel LoadedObjModel GlossColor.Color
    | ...
```

3D: not-gloss

```
play :: Real b ⇒
  Options
    → Double    -- sample time
    → world    -- initial state
    → (world → (VisObject b, Maybe Cursor))    -- draw function
    → (Float → world → world)    -- state propogation function
    → (world → IO ())    -- set where camera looks
    → Maybe (world → Key → KeyState → Modifiers → Position → world)
    → Maybe (world → Position → world)    -- mouse drag
    → Maybe (world → Position → world)    -- mouse move
    → IO ()
```

Debugování, testování, benchmarking

Jak debugovat Haskell?

Problém: koukání debuggerem na proměnné a postup výpočtu nedává smysl.

- To je ve skutečnosti dobře.
- Proměnné neexistují a hodnoty většinou nemají hodnotu (jen thunk).
- Postup výpočtu je pro člověka obtížně stravitelný
- Skompilovaný kód nemá s původním nutně společnou strukturu

Jak debugovat Haskell?

Problém: koukání debuggerem na proměnné a postup výpočtu nedává smysl.

- To je ve skutečnosti dobře.
- Proměnné neexistují a hodnoty většinou nemají hodnotu (jen thunk).
- Postup výpočtu je pro člověka obtížně stravitelný
- Skompilovaný kód nemá s původním nutně společnou strukturu

Místo toho:

- Máme efektivní interpreter
- Máme užitečný typový systém
- Umíme trasovat

Program nejde skompilovat kvůli typům

Běžný programátor nemá v hlavě typový systém. (Matfyzáci by ale měli.)

Nejlepší řešení: _ a.k.a. Typed Hole

$$\_\ (+1)\ [1, 2, 3]$$

Found hole: $\_ :: (\mathit{Integer} \to \mathit{Integer}) \to [\mathit{Integer}] \to t$

Program nejde skompilovat kvůli typům

Běžný programátor nemá v hlavě typový systém. (Matfyzáci by ale měli.)

Nejlepší řešení: _ a.k.a. Typed Hole

$$\_\ (+1)\ [1, 2, 3]$$

Found hole: _ :: (*Integer* → *Integer*) → [*Integer*] → *t*

$$\textit{foldl}\ \_\ 0\ [\texttt{"asd"}, \texttt{"qwe"}]$$

Found hole:

Program nejde skompilovat kvůli typům

Běžný programátor nemá v hlavě typový systém. (Matfyzáci by ale měli.)

Nejlepší řešení: _ a.k.a. Typed Hole

$$\_\ (+1)\ [1,2,3]$$

Found hole: _ :: (*Integer* → *Integer*) → [*Integer*] → *t*

$$\mathit{foldl}\ \_\ 0\ [\texttt{"asd"}, \texttt{"qwe"}]$$

Found hole:_ :: *b* → [*Char*] → *b*
Where: *b* is a rigid type variable with inferred type *Num b* ⇒ *b*

Užitečná pomůcka: Hoogle umi type search.

Typová chyba je moc složitá

Běžný problém: Chyba se propaguje do jiné funkce a problém způsobí až tam.

```
let concatMonoids = foldr mempty (◇)
 ...
print $ concatMonoids ["neco", "tamto"]
```

Typová chyba je moc složitá

Běžný problém: Chyba se propaguje do jiné funkce a problém způsobí až tam.

$$\textbf{let } \mathit{concatMonoids} = \mathit{foldr}\ \mathit{mempty}\ (\diamond)$$
$$\ldots$$
$$\mathit{print}\ \$\ \mathit{concatMonoids}\ [\texttt{"neco"}, \texttt{"tamto"}]$$

No instance for $(\mathit{Show}\ (\mathit{mO} \to \mathit{mO} \to \mathit{mO}))$?

Typová chyba je moc složitá

```
let concatMonoids = foldr mempty (◇)
 ...
print $ (concatMonoids ["a", "bcd"] :: _)
```

Found type wildcard _ standing for *m0* → *m0* → *m0*

Typová chyba je moc složitá

```
let concatMonoids :: _
    concatMonoids = foldr mempty (◇)
 ...
print $ concatMonoids ["a", "bcd"]
```

Found type wildcard _ standing for $[a] \to m0 \to m0 \to m0$

...to úplně nesedí s očekáváním!

Typová chyba je moc složitá

$$\textbf{let } concatMonoids :: Monoid\ m \Rightarrow [m] \to m$$
$$concatMonoids = foldr\ mempty\ (\diamond)$$
$$\ldots$$
$$print\ \$\ concatMonoids\ [\texttt{"a"}, \texttt{"bcd"}]$$

Couldn't match type m with $m0 \to m0 \to m0$
In the expression: $foldr\ mempty\ (\diamond)$

Chybová hláška nejde hlouběji, problém bude ve *foldr*

```
:t foldr
```

$$foldr :: Foldable\ t \Rightarrow (a \to b \to b) \to b \to t\ a \to b$$

Typová chyba je moc složitá

Po opravách se můžeme podívat, jestli si typový systém nemyslí něco generičtějšího:

```
:t foldr (<>) mempty
```

$$foldr\ (\diamond)\ mempty :: (Monoid\ b, Foldable\ t) \Rightarrow t\ b \rightarrow b$$

Trace

Co když potřebujeme rychle vidět, co si program myslí uprostřed nějakého výpočtu?

- C-style řešení: `printf`
- Haskell, SW development řešení:
  Všechny funkce předělám na IO, aby fungoval *print*!
- Haskell, vhodné dočasné neinvazivní řešení: *Debug*.*Trace*

Trace

Co když potřebujeme rychle vidět, co si program myslí uprostřed nějakého výpočtu?

- C-style řešení: `printf`
- Haskell, SW development řešení:
  Všechny funkce předělám na IO, aby fungoval *print*!
- Haskell, vhodné dočasné neinvazivní řešení: *Debug*.*Trace*

trace :: *String* $\rightarrow$ *a* $\rightarrow$ *a*

```
ghci> if 3 < 4 then trace "funguje" 5 else 6
funguje
5
```

(Vedlejší efekty a strictness můžou rozbít zbytek programu, v produkčním kódu nemají co dělat.)

Trace

$$trace :: String \to val \to val$$
$$traceShow :: Show\ msg \Rightarrow \to msg \to val \to val$$
$$traceShowId :: Show\ val \Rightarrow val \to val$$

$$traceM :: Applicative\ f \Rightarrow String \to f\ ()$$
$$traceShowM :: (Show\ msg, Applicative\ f) \Rightarrow msg \to f\ ()$$

import *Debug.SimpleReflect*

product $[1\,..\,10] :: \mathit{Expr}$
$\quad \rightsquigarrow 1 * 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$

foldr $(+)\ 0\ [1\,..\,4] :: \mathit{Expr}$
$\quad \rightsquigarrow 1 + (2 + (3 + (4 + 0)))$

map f $[1, 2, 3] :: [\mathit{Expr}]$
$\quad \rightsquigarrow [f\ 1, f\ 2, f\ 3]$

scanl f $0\ [1\,..\,3] :: [\mathit{Expr}]$
$\quad \rightsquigarrow [0, f\ 0\ 1, f\ (f\ 0\ 1)\ 2, f\ (f\ (f\ 0\ 1)\ 2)\ 3]$

mapM_ print $ reduction $(1 * 2 + 3 * 4)$
$\quad \rightsquigarrow 1 * 2 + 3 * 4$
$\qquad\ \ 2 + 3 * 4$
$\qquad\ \ 2 + 12$
$\qquad\ \ 14$

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

Unit testy

Unit testy nejsou v Haskellu populární, většinou jde o (zlo)zvyk z jiných jazyku.

- ☹ test-driven development je nahrazený type-driven developmentem
- ☹ absence vedlejších efektů dovoluje i obrovské programy testovat v `ghci`
- ☹ unit testy nejsou integrační testy
- ☹ fungující polymorfismus zmenšuje pokrytelný prostor
- ☼ správně napsané unit testy jdou považovat za vyšší variantu dokumentace
- ☼ v týmovém prostředí je unit testing univerzální ochranou proti ostatním programátorům

HUnit

Běžné použití:

- `cabal` nastavíme na používání unit testů
- do `test/` vyrobíme samostatný program který spouští testování
- Použujeme *Test.HUnit* k vyrobení pěkného interface programu

test/main.hs

```
import FindIdentifier (findIdentifier)
import Test.HUnit

testEmpty = TestCase $ assertEqual "Should get Nothing from an empty string"
  Nothing (findIdentifier "" (1, 1))

testNegCursor = TestCase $ assertEqual "Should get Nothing when cursor is negat
  Nothing (findIdentifier "a" (-1, -1))

testMinimal = TestCase $ assertEqual "Minimal program"
  (Just "main") (findIdentifier "main = print 42" (1, 2))

main = runTestTT $ TestList [testEmpty, testNegCursor, testMinimal]
```

QuickCheck

Psát testy ručně je otrava. Použijme fuzzer.

import *Test.QuickCheck*

$$prop\_reverse :: [Int] \to Bool$$
$$prop\_reverse\ xs = reverse\ (reverse\ xs) \equiv xs$$

```
ghci> quickCheck prop_reverse
 >>> quickCheck prop_reverse
 +++ OK, passed 100 tests.
```

QuickCheck

```
ghci> quickCheck $ \a -> a + 1 == a + 1
+++ OK, passed 100 tests.
ghci> quickCheck $ \a -> abs (a*a) == a*a
+++ OK, passed 100 tests.

ghci> quickCheck $ \a -> abs (a*a*a) == a*a*a
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
-1
```

```
ghci> quickCheck $
  \a -> (if a == 20 then 666 else a) == a
+++ OK, passed 100 tests.

ghci> quickCheck . withMaxSuccess 10000 $
  \a -> (if a == 20 then 666 else a) == a
*** Failed! Falsifiable (after 22 tests):
20
```

```
ghci> quickCheck $ \xs n -> xs !! n == head (drop n xs)
*** Failed! Exception: 'Prelude.!!: index too large'
[]
0

ghci> quickCheck $ \(NonEmpty xs) (NonNegative n) ->
        xs !! n == head (drop n xs)
*** Failed! Exception: 'Prelude.!!: index too large'
NonEmpty {getNonEmpty = [()]}
NonNegative {getNonNegative = 1}
```

```
ghci> quickCheck $ \(NonEmpty xs) (NonNegative n) ->
        n < length xs ==> xs !! n == head (drop n xs)

+++ OK, passed 100 tests; 61 discarded.
```

Jak QuickCheck hledá, co testovat?

Jak QuickCheck hledá, co testovat?

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

- *Gen* je *State*-like monáda poskytující náhodná čísla
- *arbitrary* by mělo v *Gen* výpočtu vyrobit náhodný objekt typu *a*
- *shrink* by mělo z většího generovaného náhodného objektu vyrobit pod-objekty (např. kvůli přesnější lokalizaci chyby)

Benchmarking

Benchmarky typicky vyžadují:

- spustit víc běhů
- změřit výsledek na různých velikostech dat
- neměřit okamžité líné vyhodnocování

Haskellí řešení: *Criterion* + *DeepSeq*

Criterion — demo

Zjistíme, o kolik je *Integer* pomalejší než *Int*.

$$sumN\ n = sum\ [1\,..\,n]$$
$$fact\ n = product\ [1\,..\,n]$$

$$sumNi = sumN :: Int \to Int$$
$$sumNhi = sumN :: Integer \to Integer$$
$$facti = fact :: Int \to Int$$
$$facthi = fact :: Integer \to Integer$$

Criterion — demo

Zjistíme, o kolik je *Integer* pomalejší než *Int*.

$$
\begin{aligned}
&\mathit{sumN}\ n = \mathit{sum}\ [1\,..\,n] \\
&\mathit{fact}\ n = \mathit{product}\ [1\,..\,n] \\
&\mathit{sumNi} = \mathit{sumN} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{sumNhi} = \mathit{sumN} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{facti} = \mathit{fact} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{facthi} = \mathit{fact} :: \mathit{Integer} \to \mathit{Integer}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{problemSizes} = [1, 30\,..\,100] \\
&\mathit{mkBench}\ f\ \mathit{size} = \mathit{bench}\ (\mathit{show}\ \mathit{size})\ \$\ \mathit{nf}\ f\ \$\ \mathit{fromInteger}\ \mathit{size} \\
&\mathit{mkBenches}\ f = \mathit{map}\ (\mathit{mkBench}\ f)\ \mathit{problemSizes}
\end{aligned}
$$

Criterion — demo

Zjistíme, o kolik je *Integer* pomalejší než *Int*.

$$sumN\ n = sum\ [1\,..\,n]$$
$$fact\ n = product\ [1\,..\,n]$$

$$sumNi = sumN :: Int \to Int$$
$$sumNhi = sumN :: Integer \to Integer$$
$$facti = fact :: Int \to Int$$
$$facthi = fact :: Integer \to Integer$$

$$problemSizes = [1, 30\,..\,100]$$

$$mkBench\ f\ size = bench\ (show\ size)\ \$\ nf\ f\ \$\ fromInteger\ size$$
$$mkBenches\ f = map\ (mkBench\ f)\ problemSizes$$

Criterion — demo

Zjistíme, o kolik je *Integer* pomalejší než *Int*.

$$\mathit{sumN}\ n = \mathit{sum}\ [1\,..\,n]$$
$$\mathit{fact}\ n = \mathit{product}\ [1\,..\,n]$$

$$\mathit{sumNi} = \mathit{sumN} :: \mathit{Int} \to \mathit{Int}$$
$$\mathit{sumNhi} = \mathit{sumN} :: \mathit{Integer} \to \mathit{Integer}$$
$$\mathit{facti} = \mathit{fact} :: \mathit{Int} \to \mathit{Int}$$
$$\mathit{facthi} = \mathit{fact} :: \mathit{Integer} \to \mathit{Integer}$$

$$\mathit{problemSizes} = [1, 30\,..\,100]$$

$$\mathit{mkBench}\ f\ \mathit{size} = \mathit{bench}\ (\mathit{show}\ \mathit{size})\ \$\ \mathit{nf}\ f\ \$\ \mathit{fromInteger}\ \mathit{size}$$
$$\mathit{mkBenches}\ f = \mathit{map}\ (\mathit{mkBench}\ f)\ \mathit{problemSizes}$$

Criterion — demo

```
main = defaultMain [
  bgroup "sum-int" $ mkBenches sumNi,
  bgroup "sum-integer" $ mkBenches sumNhi,
  bgroup "fact-int" $ mkBenches facti,
  bgroup "fact-integer" $ mkBenches facthi]
```

Hint: je vhodné kompilovat s `-O`

Criterion — demo

```
main = defaultMain [
  bgroup "sum-int" $ mkBenches sumNi,
  bgroup "sum-integer" $ mkBenches sumNhi,
  bgroup "fact-int" $ mkBenches facti,
  bgroup "fact-integer" $ mkBenches facthi]
```

Hint: je vhodné kompilovat s `-O`

Criterion — demo

```
$ ./ints --help
Microbenchmark suite - built with criterion 1.5.2.0

Usage: ints [-I|--ci CI] [-L|--time-limit SECS]
            [--resamples COUNT] ...

$ ./ints
benchmarking sum-int/1
time                 8.832 ns   (8.723 ns .. 8.952 ns)
                     0.996 R2   (0.991 R2 .. 0.999 R2)
mean                 9.068 ns   (8.829 ns .. 9.675 ns)
std dev              1.246 ns   (536.7 ps .. 2.682 ns)
variance introduced by outliers: 96% (severely inflated)
...
```

Criterion — výsledek

```
benchmarking sum-int/88
time                 75.10 ns   (74.46 ns .. 76.07 ns)

benchmarking sum-integer/88
time                 1.529 us   (1.516 us .. 1.550 us)

benchmarking fact-int/88
time                 103.7 ns   (99.13 ns .. 108.3 ns)

benchmarking fact-integer/88
time                 3.645 us   (3.593 us .. 3.723 us)
```

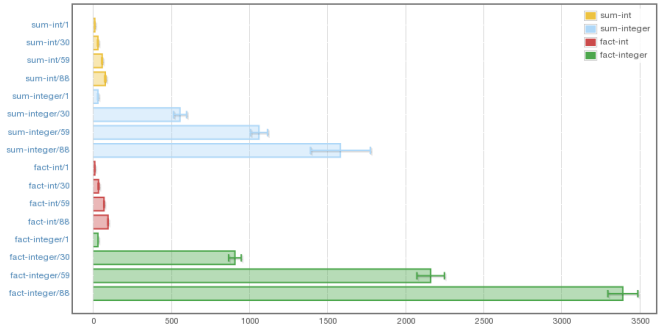
Criterion — výsledek

```
$ ./ints --output vysledek.html
```

criterion performance measurements

overview

want to understand this report?

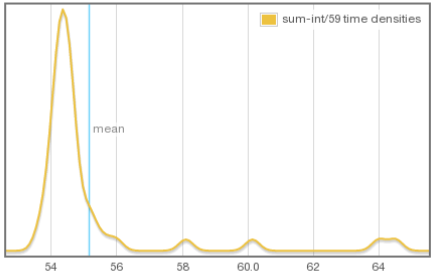


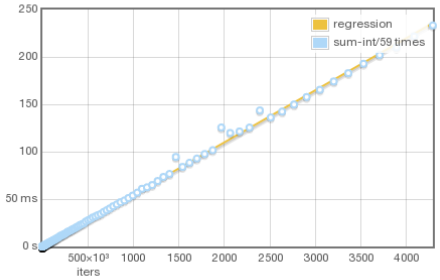
sum-int/1

Criterion — výsledek

```
$ ./ints --output vysledek.html
```

sum-int/59





	lower bound	estimate	upper bound
OLS regression	54.5 ns	**55.0 ns**	55.7 ns
R^2 goodness-of-fit	0.997	**0.998**	1.000
Mean execution time	54.7 ns	**55.2 ns**	56.2 ns
Standard deviation	1.22 ns	**2.30 ns**	3.69 ns

Outlying measurements have severe (63.5%) effect on estimated standard deviation.

`nf` na vlastním typu

Problém: není genericky jasné, jak (a jak moc) forcovat složitější typy.

```
nf :: Control.DeepSeq.NFData b ⇒
      (a → b) → a → Benchmarkable

data Tree = Leaf Int | Branch Tree Tree
instance NFData Tree where
  rnf (Leaf i)       = rnf i `seq` ()
  rnf (Branch l r) = rnf l `seq` rnf r `seq` ()
```

Definice *seq* nesouvisí s pořadím vyhodnocení!

`nf` na vlastním typu

Problém: není genericky jasné, jak (a jak moc) forcovat složitější typy.

```
nf :: Control.DeepSeq.NFData b ⇒
      (a → b) → a → Benchmarkable

data Tree = Leaf Int | Branch Tree Tree
instance NFData Tree where
  rnf (Leaf i)       = rnf i `seq` ()
  rnf (Branch l r) = rnf l `seq` rnf r `seq` ()
```

Definice *seq* nesouvisí s pořadím vyhodnocení!

Pokud *a* má slabou hlavově normální formu (WHNF), *seq a b* vrátí *b*. Pokud *a* nemá WHNF, *seq a b* taky nemá WHNF.

(intuice: *min a b*)

Webové aplikace

Architektura typické webové aplikace

- Databáze
  - CokolivSQL ElasticSearch, Mongo, Redis, …
- API klient
- Back-end
  - webserver, request routing, API endpointy
- API
  - HTML
  - JSON
  - …
- Front-end
  - Javascript
- ~~Browser~~

Haskell jde nasadit na všech úrovních.

Architektura typické webové aplikace

- Databáze
  - CokolivSQL ElasticSearch, Mongo, Redis, …
- API klient
- Back-end
  - webserver, request routing, API endpointy
- API
  - HTML
  - JSON
  - …
- Front-end
  - Javascript
- ~~Browser~~

Haskell jde nasadit na všech úrovních.

HTTP klient

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "https://test.cz/stranka"
  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP klient

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "https://test.cz/stranka"
  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP klient

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "https://test.cz/stranka"
  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP klient

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "https://test.cz/stranka"
  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

API klient

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://data.cz/vec"
  let request
        = setRequestMethod "PUT"
        $ setRequestPath "/polozit"
        $ setRequestQueryString [("parametr", Just "hodnota")]
        $ setRequestBodyLBS "DataData"
        $ setRequestSecure True
        $ setRequestPort 443
        $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API klient

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://data.cz/vec"
  let request
        = setRequestMethod "PUT"
        $ setRequestPath "/polozit"
        $ setRequestQueryString [("parametr", Just "hodnota")]
        $ setRequestBodyLBS "DataData"
        $ setRequestSecure True
        $ setRequestPort 443
        $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API klient

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' <- parseRequest "POST http://data.cz/vec"
  let request
        = setRequestMethod "PUT"
        $ setRequestPath "/polozit"
        $ setRequestQueryString [("parametr", Just "hodnota")]
        $ setRequestBodyLBS "DataData"
        $ setRequestSecure True
        $ setRequestPort 443
        $ request'
  response <- httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API klient

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://data.cz/vec"
  let request
        = setRequestMethod "PUT"
        $ setRequestPath "/polozit"
        $ setRequestQueryString [("parametr", Just "hodnota")]
        $ setRequestBodyLBS "DataData"
        $ setRequestSecure True
        $ setRequestPort 443
        $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

JSON

Standardní řešení parsování a generování JSONu je *Data.Aeson*.

```
data Value = Object (Map Text Value)
           | Array (Vector Value)
           | String Text | Number Number | Bool Bool
           | Null

class FromJSON a where
  parseJSON :: Value → Parser a

class ToJSON a where
  toJSON :: a → Value
```

```
data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}

instance ToJSON Hodnota where
  toJSON (Hodnota m j) = Object $
    fromList [("mnozstvi", Number (D m)),
              ("jednotka", String j)]

instance FromJSON Hodnota where
  parseJSON (Object a) =
    Hodnota <$> a .: "mnozstvi"
            <*> a .: "jednotka"
  parseJSON _ = mzero
```

```
data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}

instance ToJSON Hodnota where
  toJSON (Hodnota m j) = Object $
    fromList [("mnozstvi", Number (D m)),
              ("jednotka", String j)]

instance FromJSON Hodnota where
  parseJSON (Object a) =
    Hodnota <$> a .: "mnozstvi"
            <*> a .: "jednotka"
  parseJSON _ = mzero
```

```
data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}

instance ToJSON Hodnota where
  toJSON (Hodnota m j) = Object $
    fromList [("mnozstvi", Number (D m)),
              ("jednotka", String j)]

instance FromJSON Hodnota where
  parseJSON (Object a) =
    Hodnota <$> a .: "mnozstvi"
            <*> a .: "jednotka"
  parseJSON _ = mzero
```

```
data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}

instance ToJSON Hodnota where
  toJSON (Hodnota m j) = Object $
    fromList [("mnozstvi", Number (D m)),
              ("jednotka", String j)]

instance FromJSON Hodnota where
  parseJSON (Object a) =
    Hodnota <$> a .: "mnozstvi"
            <*> a .: "jednotka"
  parseJSON _ = mzero
```

Výhoda: prakticky všechny JSONovité balíky umí transparentně používat instance *FromJSON* a *ToJSON*; běžné typy mají instance předpřipravené.

Manuální použití:

$$\begin{aligned}
&\mathit{encode} :: \mathit{ToJSON}\ a \Rightarrow a \to \mathit{ByteString} \\
&\mathit{decode} :: \mathit{FromJSON}\ a \Rightarrow \mathit{ByteString} \to \mathit{Maybe}\ a
\end{aligned}$$

$$\begin{aligned}
&\mathit{decode}\ \texttt{"[123,234]"} :: \mathit{Maybe}\ [\mathit{Int}] \\
&\qquad \rightsquigarrow \mathit{Just}\ [123, 234] \\
&\mathit{encode}\ [123, 234] \\
&\qquad \rightsquigarrow \texttt{"[123,234]"}
\end{aligned}$$

Aeson automaticky

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}
  deriving (Show, Generic)

instance ToJSON Hodnota where
  toJSON = genericToJSON

instance FromJSON Hodnota where
  parseJSON = genericParseJSON
```

```
decode "{ \"mnozstvi\": 123, \"jednotka\": \"cm\" }" :: Maybe Hodnota
    ⇝ Just (Hodnota {mnozstvi = 123.0, jednotka = "cm"})
encode $ Hodnota 42 "kg"
    ⇝ "{\"jednotka\":\"kg\",\"mnozstvi\":42}"
```

Aeson automaticky

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}
  deriving (Show, Generic)

instance ToJSON Hodnota where
  toJSON = genericToJSON

instance FromJSON Hodnota where
  parseJSON = genericParseJSON

decode "{ \"mnozstvi\": 123, \"jednotka\": \"cm\" }" :: Maybe Hodnota
    ⇝ Just (Hodnota {mnozstvi = 123.0, jednotka = "cm"})
encode $ Hodnota 42 "kg"
    ⇝ "{\"jednotka\":\"kg\",\"mnozstvi\":42}"
```

Aeson automaticky

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Hodnota = Hodnota {mnozstvi :: Double,
                        jednotka :: String}
  deriving (Show, Generic)

instance ToJSON Hodnota where
  toJSON = genericToJSON

instance FromJSON Hodnota where
  parseJSON = genericParseJSON

decode "{ \"mnozstvi\": 123, \"jednotka\": \"cm\" }" :: Maybe Hodnota
    ⇝ Just (Hodnota {mnozstvi = 123.0, jednotka = "cm"})
encode $ Hodnota 42 "kg"
    ⇝ "{\"jednotka\":\"kg\",\"mnozstvi\":42}"
```

Aeson automaticky s customizací

Customizace pomocí *defaultOptions* :: *Options*:

```
instance ToJSON Hodnota where
  toJSON = genericToJSON $ defaultOptions
    { fieldLabelModifier = λs → "hodnota_" ◇ s }
encode $ Hodnota 42 "kg"
    ⇝ "{\"hodnota_jednotka\":\"kg\",\"hodnota_mnozstvi\":42}"
```

Options jde obdobně použít i pro *FromJSON*.

Aeson automaticky s customizací

Customizace pomocí *defaultOptions* :: *Options*:

```
instance ToJSON Hodnota where
  toJSON = genericToJSON $ defaultOptions
    { fieldLabelModifier = λs → "hodnota_" ◇ s }
encode $ Hodnota 42 "kg"
    ⇝ "{\"hodnota_jednotka\":\"kg\",\"hodnota_mnozstvi\":42}"
```

Options jde obdobně použít i pro *FromJSON*.

Aeson automaticky s customizací

Customizace pomocí *defaultOptions* :: *Options*:

```
instance ToJSON Hodnota where
  toJSON = genericToJSON $ defaultOptions
    { fieldLabelModifier = λs → "hodnota_" ◇ s }
encode $ Hodnota 42 "kg"
    ⇝ "{\"hodnota_jednotka\":\"kg\",\"hodnota_mnozstvi\":42}"
```

Options jde obdobně použít i pro *FromJSON*.

HTML

Generovat HTML Haskellem ručně je poměrně jednoduché. Dostupné knihovny jako Blaze a Lucid minimalizují overhead slepování různých druhů stringů.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

HTML

Generovat HTML Haskellem ručně je poměrně jednoduché. Dostupné knihovny jako Blaze a Lucid minimalizují overhead slepování různých druhů stringů.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Většina kombinátorů akceptuje ‘obsah’, monády jsou použité jako builder.

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

HTML

Generovat HTML Haskellem ručně je poměrně jednoduché. Dostupné knihovny jako Blaze a Lucid minimalizují overhead slepování různých druhů stringů.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Většina kombinátorů akceptuje ‘obsah’, monády jsou použité jako builder.

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

HTML

Generovat HTML Haskellem ručně je poměrně jednoduché. Dostupné knihovny jako Blaze a Lucid minimalizují overhead slepování různých druhů stringů.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Většina kombinátorů akceptuje ‘obsah’, monády jsou použité jako builder.

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

Atributy:

```
image :: Html ()
image = img_
  [class_ "styled"
  ,src_ "lambda.png"
  ,alt_ "An image of lambda."
  ,id_ "theimage"
  ]
```

Atributy:

```
image :: Html ()
image = img_
  [class_ "styled"
  ,src_ "lambda.png"
  ,alt_ "An image of lambda."
  ,id_ "theimage"
  ]
```

Obsah tagu:

```
odstavec :: Html ()
odstavec = p_ [class_ "zarovnat"] $ em_ "Věta!"
```

SQLite overview

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket c = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

SQLite overview

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket c = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

SQLite overview

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket c = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

SQL DSL (Selda)

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)
instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic
instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)
instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic
instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)
instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic
instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"      19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"        10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name : ∗ : person ! #pet
  liftIO $ print adultsAndTheirPets
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"      19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"        10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name : ∗ : person ! #pet
  liftIO $ print adultsAndTheirPets
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"      19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"        10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name : ∗ : person ! #pet
  liftIO $ print adultsAndTheirPets
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"      19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"        10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name : ∗ : person ! #pet
  liftIO $ print adultsAndTheirPets
```

HTTP Back-end

Nejběžnější balíky:

- `wai` — Web Application Interface, nízkoúrovňový základ pro HTTP protokol
- `warp` — HTTP server
- *Yesod* — `warp` obalený RESTful datovým modelem a Template Haskellem
- *Scotty* — podstatně tenčí a praktičtější obal
- *Servant* — Typové definice API, automatické generování serverového, klientského a javascriptového kódu.

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types
import Network.Wai.Handler.Warp (run)

app :: Application
app _ respond = do
  putStrLn "Někdo něco chce!"
  respond $ responseLBS
    status200
    [("Content-Type", "text/html")]
    "<html><body><h1>Nazdar!</h1></body></html>"

main = run 8080 app
```

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types
import Network.Wai.Handler.Warp (run)

app :: Application
app _ respond = do
  putStrLn "Někdo něco chce!"
  respond $ responseLBS
    status200
    [("Content-Type", "text/html")]
    "<html><body><h1>Nazdar!</h1></body></html>"

main = run 8080 app
```

Scotty

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/pozdrav/:jmeno" $ do
    jmeno ← param "jmeno"
    html $ "<h1>Nazdar, " ◇ jmeno ◇ "!</h1>"
  post "/log/:zprava" $ do
    param "zprava" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

Scotty

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/pozdrav/:jmeno" $ do
    jmeno ← param "jmeno"
    html $ "<h1>Nazdar, " ◇ jmeno ◇ "!</h1>"
  post "/log/:zprava" $ do
    param "zprava" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

Scotty

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/pozdrav/:jmeno" $ do
    jmeno ← param "jmeno"
    html $ "<h1>Nazdar, " ◇ jmeno ◇ "!</h1>"
  post "/log/:zprava" $ do
    param "zprava" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

Scotty

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/pozdrav/:jmeno" $ do
    jmeno ← param "jmeno"
    html $ "<h1>Nazdar, " ◇ jmeno ◇ "!</h1>"
  post "/log/:zprava" $ do
    param "zprava" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

Scotty

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/pozdrav/:jmeno" $ do
    jmeno ← param "jmeno"
    html $ "<h1>Nazdar, " ◇ jmeno ◇ "!</h1>"
  post "/log/:zprava" $ do
    param "zprava" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeOperators #-}

import Prelude ()
import Prelude.Compat

import Control.Monad.Except
import Control.Monad.Reader
import Data.Aeson
import Data.Aeson.Types
import Data.Attoparsec.ByteString
import Data.ByteString (ByteString)
import Data.List
import Data.Maybe
import Data.String.Conversions
import Data.Time.Calendar
import GHC.Generics
import Lucid
import Network.HTTP.Media ((//), (/:))
import Network.Wai
import Network.Wai.Handler.Warp
import Servant
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

Servant

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

Jak se zeptat na API?

```
users :<|> posts = client userAPI
```

Použití:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "userdb.cz" 8080 "")
  )
  :: IO [User]
```

Jak se zeptat na API?

```
users : ⟨|⟩ posts = client userAPI
```

Použití:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "userdb.cz" 8080 "")
  )
  :: IO [User]
```

Jak se zeptat na API?

```
users : ⟨|⟩ posts = client userAPI
```

Použití:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "userdb.cz" 8080 "")
  )
  :: IO [User]
```

Jak se zeptat na API?

```
users : ⟨|⟩ posts = client userAPI
```

Použití:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "userdb.cz" 8080 "")
  )
  :: IO [User]
```

Bonusy!

Javaskriptový API klient pro jQuery:

$$apiJS :: Text$$
$$apiJS = jsForAPI\ userAPI\ jquery$$

API dokumentující jiné API:

$$apiDocs :: API$$
$$apiDocs = docs\ userAPI$$

Front-end

Tradiční webový front-endový jazyk je Javaskript (a moc toho s tím nenaděláme).

Front-end

Tradiční webový front-endový jazyk je Javaskript (a moc toho s tím nenaděláme).

Haskell jde přeložit do Javaskriptu!

- GHCJS (`https://github.com/ghcjs/ghcjs`)
- Existuje několik užitečných knihoven, např. Reflex-DOM.

Méně brutální řešení:

- (Coffee/Type)Script
- PureScript
- Elm

(jazyky jsou občas polovičaté, ale pořád lepší než Javaskript)

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view }

type Msg = Increment | Decrement
```

```
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [button [onClick · (Decrement)] [text "-"]
    , div [] [text (String.fromInt model)]
    , button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view }

type Msg = Increment | Decrement
```

```
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [button [onClick · (Decrement)] [text "-"]
    , div [] [text (String.fromInt model)]
    , button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view }

type Msg = Increment | Decrement
```

```
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [button [onClick · (Decrement)] [text "-"]
    , div [] [text (String.fromInt model)]
    , button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [button [onClick · (Decrement)] [text "-"]
    , div [] [text (String.fromInt model)]
    , button [onClick · (Increment)] [text "+"]
    ]
```

Náhled na FRP a Reflex-DOM

Funkcionální Reaktivní Programování je zajímavý způsob jak popisovat dynamicky se měnící objekty. Uživatelské interfacy jsou vhodné místo k použití.

Náhled na FRP a Reflex-DOM

Funkcionální Reaktivní Programování je zajímavý způsob jak popisovat dynamicky se měnící objekty. Uživatelské interfacy jsou vhodné místo k použití.

- Tradičně:
  - *getMousePos*
  - *setWidgetPosition* (*mousePos*)
- Event-based: *onMousePosChange* (*setWidgetPosition*)
- Reaktivně: *widget* $\{position = mousePos\}$

FRP modeluje data programu jako chování (Behavior), což je abstrakce celého průběhu hodnot v čase běhu programu.

FRP modeluje data programu jako chování (Behavior), což je abstrakce celého průběhu hodnot v čase běhu programu.

To samozřejmě nejde. Budeme říkat, že to je ‘implementační detail’.

FRP demo

```
{-# LANGUAGE OverloadedStrings #-}
import Reflex.Dom

main = mainWidget $ el "div" $ do
  t ← inputElement "default1"
  t2 ← inputElement "default2"
  dynText $ _inputElement_value t
              ◇ ", " ◇
              _inputElement_value t2
```

(kód je zjednodušený)

```
data InputElement er d t
  = InputElement
  { _inputElement_value :: Dynamic t Text
  , _inputElement_checked :: Dynamic t Bool
  , _inputElement_checkedChange :: Event t Bool
  , _inputElement_input :: Event t Text
  , _inputElement_hasFocus :: Dynamic t Bool
  , _inputElement_element :: Element er d t
  , _inputElement_raw :: RawInputElement d
  , _inputElement_files :: Dynamic t [RawFile d]
  }
```

```
data InputElement er d t
  = InputElement
  { _inputElement_value :: Dynamic t Text
  , _inputElement_checked :: Dynamic t Bool
  , _inputElement_checkedChange :: Event t Bool
  , _inputElement_input :: Event t Text
  , _inputElement_hasFocus :: Dynamic t Bool
  , _inputElement_element :: Element er d t
  , _inputElement_raw :: RawInputElement d
  , _inputElement_files :: Dynamic t [RawFile d]
  }
```

Nějakou jednodušší variantu FRP dozajista čeká zářivá budoucnost.

eDSL a interprety

Motivace

- ADTs jsou skvělé na reprezentaci výrazů a programů
  (podíváme se na několik alternativ a pomůcek)
- embedované jazyky mají minimální overhead díky agresivním optimalizacím
- díky efektovým systémům je dost jednoduché psát kvalitní interpretery

Rozšiřitelné analyzovatelné datové struktury

```
data Expr
    = Var String
    | Lam String Expr
    | App Expr Expr
    | Plus Expr Expr
    | Negate Expr
    | ...
```

- Je string dobrý typ pro proměnné?
- Budou existující funkce fungovat když přidám jazykovou featuru?
- Je výraz (nějak) validní?
- Musím všechny funkce psát rekurzivně?
- …

Běžná pomůcka: Parametrizace typu proměnných

```
data Expr a
  = Var a
  | Lam a (Expr a)
  | App (Expr a) (Expr a)
  | Plus (Expr a) (Expr a)
  | Negate (Expr a)
  | ...
```

- většinou dává smysl jako funktor
- místo Stringů můžete použít např. indexy nebo reference (rychlost)

Higher Order Abstract Syntax

```
data Expr a where
  Val :: a → Expr a
  Lam :: (Expr a → Expr b) → Expr (a → b)
  App :: Expr (a → b) → Expr a → Expr b

id' :: Expr (a → a)
id' = Lam (λx → x)

const' :: Expr (a → b → a)
const' = Lam (λx → Lam (λy → x))

eval :: Expr a → a
eval (Val v) = v
eval (Lam f) = λx → eval (f (Val x))
eval (App f p) = (eval f) (eval p)
```

- typ výrazu je určitě správně!
- poněkud nepohodlné (parsery musí znát typ předem, prettyprinting je komplikovaný)

Problém: Struktura v ADT syntaxi neposkytuje žádné typové záruky (ale jde ‘rozebrat’ ručně), HOAS je nedostatečně patternmatchovatelný (ale je typovaný Haskellem).

Final interpreters

```
class Expr repr where
  val :: Int → repr
  add :: repr → repr → repr
  mul :: repr → repr → repr

instance Expr Int where
  val n = n
  add a b = a + b
  mul a b = a * b

instance Expr String where
  val n = show n
  add a b = "(" ++ a ++ "+" ++ b ++ ")"
  mul a b = "(" ++ a ++ "*" ++ b ++ ")"
```

```
eval :: Int → Int
eval = id

render :: String → String
render = id

expr :: Expr r ⇒ r
expr = add (val 1) (mul (val 2) (val 3))

eval expr ≡ 7
render expr ≡ "(1+(2*3))"
```

Final interpreters — výhody

Horizontální rozšiřitelnost

```
instance Expr (Writer [Asm]) where ...
compile :: Writer [Asm] → Writer [Asm]
compile = id
```

Vertikální rozšiřitelnost

```
class Expr repr ⇒ Lang repr
  var :: String → repr
  assign :: String → repr → repr
  semicolon :: repr → repr → repr
instance Lang (Writer [Asm]) where
  ...
  semicolon a b = a ≫ b
```

Jak vypadá final encoding v paměti?

Jak vypadá final encoding v paměti?

AST je vyrobené z funkcí pospojovaných v STG.

Tagless final

```
class Expr rep where
  lam :: (rep a → rep b) → rep (a → b)
  app :: rep (a → b) → (rep a → rep b)
  val :: a → rep a

newtype Interpret a = R {reify :: a}

instance Expr Interpret where
  lam f = R $ reify . f . R
  app f a = R $ reify f $ reify a
  val = R

example :: Expr rep ⇒ rep Int
example = app (lam (λx → x)) (val 3)

eval :: Interpret a → a
eval e = reify e
```

- tagged final interpretery si musí typovou informaci ukládat ručně
- v tagless verzi se o typy stará kompilátor a není potřeba je pracně ukládat a matchovat

Následuje sbírka užitečných řešení typických problémů

Anotace kódu (typicky lokace)

```
type Loc = Int
data Expr
    = Val Loc Int
    | Plus Loc Expr Expr
```

Anotace kódu (typicky lokace)

```
type Loc = Int
data Expr
  = Val Loc Int
  | Plus Loc Expr Expr
```

Lepší:

```
data L a = L Int a
type Expr a = L (Ex a)
data Ex a = Val Int | Plus Expr Expr
```

Anotace kódu (typicky lokace)

```
type Loc = Int
data Expr
  = Val Loc Int
  | Plus Loc Expr Expr
```

Lepší:

```
data L a = L Int a
type Expr a = L (Ex a)
data Ex a = Val Int | Plus Expr Expr
```

Systematické (*Data.Fix*):

```
newtype Fix f = Fix {unFix :: f (Fix f)}
data Expr f = Val Int | Plus f f
data Loc e f = Loc Int (e f)
type LocExpr = Fix (Loc Expr)
loc l e = Fix (Loc l e)
loc 0 (Val 3) :: LocExpr
```

Výhoda: při změnách struktury (např.: odmazání lokací) není potřeba vyrábět nový typ a nové funkce, stačí použít *Fix Expr*.


Vorsicht
Funktor
2 m Abstand halten

Monadické DSL bez monád

Monády jdou vyrábět z funktorů zadarmo
(*Control.Monad.Free*):

```
data Free f a
  = Pure a
  | Free (f (Free f a))

instance Functor f ⇒ Monad (Free f) where
  return = Pure
  Pure a >>= f = f a
  Free m >>= f = Free (fmap (>>=f) m)
```

Monadické DSL bez monád

Monády jdou vyrábět z funktorů zadarmo (*Control.Monad.Free*):

```
data Free f a
  = Pure a
  | Free (f (Free f a))

instance Functor f ⇒ Monad (Free f) where
  return = Pure
  Pure a >>= f = f a
  Free m >>= f = Free (fmap (>>=f) m)
```

- *Pure* hodnoty se chovají ‘normálně’
- Každé použití *Free* vyrobí jednu (později interpretovatelnou) vrstvu funktoru

```
Just 3 >> Nothing
  ⇝ Nothing

Free (Just $ pure 3) >> Free Nothing
  ⇝ Free (Just (Free Nothing))

Pure 123 >>= λx → return x
  ⇝ Free (Just (Pure 123))

Free (Just (Pure 1)) >> Free (Just (Pure 3))
  ⇝ Free (Just (Free (Just (Pure 3))))

sum ⟨$⟩ traverse (λx → Free (x, pure x)) [1 .. 5]
  ⇝ Free (1, Free (2, Free (3, Free (4, Free (5,
        Pure 15)))))
```

Volné monády — použití

```
data ConsoleCmd a
  = WriteLn String a
  | ReadLn (String → a)
  deriving Functor

type ConsoleM = Free ConsoleCmd

liftF :: Functor f ⇒ f a → Free f a
liftF = Free · fmap Pure

writeLn s = liftF (WriteLn s ())
readLn = liftF (ReadLn id)

program =
  x ← readLn
  y ← readLn
  writeLn (show $ read x + read y)
```

Volné monády — použití

```
data ConsoleCmd a
  = WriteLn String a
  | ReadLn (String → a)
  deriving Functor

type ConsoleM = Free ConsoleCmd

liftF :: Functor f ⇒ f a → Free f a
liftF = Free · fmap Pure

writeLn s = liftF (WriteLn s ())
readLn = liftF (ReadLn id)

program =
  x ← readLn
  y ← readLn
  writeLn (show $ read x + read y)
```

```
consInIO :: ConsoleCmd a → IO a
consInIO (WriteLn s cont) = do
  putStrLn s
  pure cont
consInIO (ReadLn callback) = do
  s ← getLine
  pure (callback s)

-- Control.Monad.Free
foldFree :: Monad m
         ⇒ (∀x. f x → m x)
         → Free f a → m a

main = foldFree consInIO program
```

Interpret jde jednoduše vyměnit, např. *consInMock*, *consInConduit*, …

Selective

- Aplikativní funktory umí popsat i poměrně složité procesy, ale nejde v nich rozhodovat o ‘tvaru’ výpočtu (a o vedlejších efektech) na základě mezivýsledků.
- Monády umí $\gg=$, ale ke statické analýze je nutné je spustit a interpretovat.

Selective

- Aplikativní funktory umí popsat i poměrně složité procesy, ale nejde v nich rozhodovat o ‘tvaru’ výpočtu (a o vedlejších efektech) na základě mezivýsledků.
- Monády umí $\gg=$, ale ke statické analýze je nutné je spustit a interpretovat.

class *Applicative f* $\Rightarrow$ *Selective f* **where**
  select :: *f* (*Either a b*) $\rightarrow$ *f* (*a* $\rightarrow$ *b*) $\rightarrow$ *f b*

ifS :: *Selective f* $\Rightarrow$ *f Bool* $\rightarrow$ *f a* $\rightarrow$ *f a* $\rightarrow$ *f a*

Výhoda: Vhodným funktorem je možné např. posbírat všechny možné chyby.
(podobně: `optparse-applicative`)

Selective

- Aplikativní funktory umí popsat i poměrně složité procesy, ale nejde v nich rozhodovat o ‘tvaru’ výpočtu (a o vedlejších efektech) na základě mezivýsledků.
- Monády umí $\gg=$, ale ke statické analýze je nutné je spustit a interpretovat.

```
class Applicative f ⇒ Selective f where
  select :: f (Either a b) → f (a → b) → f b

ifS :: Selective f ⇒ f Bool → f a → f a → f a
```

Výhoda: Vhodným funktorem je možné např. posbírat všechny možné chyby.
(podobně: `optparse-applicative`)

```
data Shape = Circle Int | Rectangle Int Int

shape :: Selective f ⇒
         f Bool → f Int → f Int → f Int → f Shape
shape s r w h = ifS s (Circle ⟨$⟩ r) (Rectangle ⟨$⟩ w ⟨*⟩ h)

shape (Success True) (Success 10)
                     (Failure ["no width"])
                     (Failure ["no height"])
  ⇝ Success (Circle 10)

shape (Success False) (Failure ["no radius"])
                      (Failure ["no width"])
                      (Failure ["no height"])
  ⇝ Failure ["no width", "no height"]
```

✎Mokhov, A., Lukyanov, G., Marlow, S., & Dimino, J. (2019). *Selective applicative functors.* Proceedings of the ACM on Programming Languages, 3(ICFP), 1-29.

Take-home message

$$\text{Functor} < \text{Applicative} \leqslant \text{Selective} < \text{Monad}$$

Teoretická odbočka: vyšší pohled na interpretery a kompilátory

Částečným vyhodnocovač kódu je program, který ze zdrojového kódu programu obsahujícího statickou (známou) a dynamickou (neznámou) část vyrobí kód programu neobsahující staticky odvoditelný výpočet.

$$\begin{aligned} mix :: & \ Program \\ & \to StaticInput \\ & \to Program \end{aligned}$$

Podobně: $s_{m,n}$ věta ve vyčíslitelnosti, *peval*, "specializace"

Futamura projections (1970):

0. Interpretace zdrojového kódu:
   interp source input $=$ *mix source input*
   (zbyde jen program který přímo vypíše výstup)
1. Kompilace programu:
   mix interp source
2. Výroba kompilátoru:
   mix mix interp
3. Univerzální překladač interpreterů na kompilátory:
   mix mix mix

Rozšiřitelnost

Programy pracující s komplikovanými datovými strukturami se typicky rozšiřují následovně:

Vertikálně Přidáním typů dat
(OOP: odvozením nových tříd, ADT: přidáním dalších možností do součtového typu, …)

Horizontálně Přidáním funkcionality
(OOP: rozšíření interfacu, ADT: přidání funkcí/instancí)

Problém: Množství kódu je $\mathcal{O}(h \cdot v)$, i triviální rozšíření vyžadují implementovat/upravit $\mathcal{O}(h + v)$ funkcí, typicky ‘boilerplate’.

Cíl: Chceme implementovat jen to zajímavé.

Jak genericky traverzovat (a modifikovat) struktury?

Nejdřív potřebujeme trochu materializovat typovou bezpečnost.

class *Typeable a* -- obsah nás nezajímá

$cast :: (Typeable\ a, Typeable\ b) \Rightarrow a \to Maybe\ b$

$(cast'\ a') :: Maybe\ Bool$

$\leadsto Nothing$

$(cast\ True) :: Maybe\ Bool$

$\leadsto Just\ True$

✎Lämmel, R., & Jones, S. P. (2003). *Scrap your boilerplate: a practical design pattern for generic programming.* ACM SIGPLAN Notices, 38(3), 26-37.

Generické transformace na neznámém typu nic neudělají:

$$
\begin{aligned}
&\mathit{mkT} :: (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \\
&\qquad \Rightarrow (a \to a) \to b \to b \\
&\mathit{mkT}\ f = \textbf{case}\ \mathit{cast}\ f\ \textbf{of} \\
&\quad \mathit{Just}\ g \to g \\
&\quad \mathit{Nothing} \to \mathit{id} \\
&\mathit{mkT}\ (\neg)\ \mathit{True} \\
&\qquad \leadsto \mathit{False} \\
&\mathit{mkT}\ (\neg)\ \texttt{'a'} \\
&\qquad \leadsto \texttt{'a'}
\end{aligned}
$$

Se znalostí ‘tvaru’ dat je generické transformace možné spouštět na celých datových typech:

```
class Typeable a ⇒ Term a where
  gmapT :: (∀b.Term b ⇒ b → b) → a → a

instance Term Either a b where
  gmapT f (Left l) = Left (f l)
  gmapT f (Right r) = Right (f r)

instance Term (a, b) where
  gmapT f (l, r) = (f l, f r)
```

Se znalostí ‘tvaru’ dat je generické transformace možné spouštět na celých datových typech:

$$\textbf{class } \mathit{Typeable}\ a \Rightarrow \mathit{Term}\ a \textbf{ where}$$
$$\quad \mathit{gmapT} :: (\forall b.\mathit{Term}\ b \Rightarrow b \to b) \to a \to a$$

$$\textbf{instance } \mathit{Term}\ \mathit{Either}\ a\ b \textbf{ where}$$
$$\quad \mathit{gmapT}\ f\ (\mathit{Left}\ l) = \mathit{Left}\ (f\ l)$$
$$\quad \mathit{gmapT}\ f\ (\mathit{Right}\ r) = \mathit{Right}\ (f\ r)$$

$$\textbf{instance } \mathit{Term}\ (a, b) \textbf{ where}$$
$$\quad \mathit{gmapT}\ f\ (l, r) = (f\ l, f\ r)$$

Se znalostí ‘tvaru’ dat je generické transformace možné spouštět na celých datových typech:

class *Typeable a* $\Rightarrow$ *Term a* **where**
  $gmapT :: (\forall b.Term\ b \Rightarrow b \to b) \to a \to a$

instance *Term Either a b* **where**
  $gmapT\ f\ (Left\ l) = Left\ (f\ l)$
  $gmapT\ f\ (Right\ r) = Right\ (f\ r)$

instance *Term* (a, b) **where**
  $gmapT\ f\ (l, r) = (f\ l, f\ r)$

$gmapT\ (mkT\ (\neg))\ (True, \texttt{'a'})$
  $\leadsto (False, \texttt{'a'})$

Rekurzivně:

$$
\begin{aligned}
&\mathit{everywhere} :: \mathit{Term}\ a \\
&\qquad \Rightarrow (\forall b.\mathit{Term}\ b \Rightarrow b \to b) \\
&\qquad \to a \to a \\
&\mathit{everywhere}\ f\ x = f\ (\mathit{gmapT}\ (\mathit{everywhere}\ f)\ x) \\
&\mathit{everywhere}\ (\mathit{mkT}\ (+1))\ (\mathit{True}, \texttt{'a'}, (1, \mathit{False}, [\mathit{True}, \mathit{False}])) \\
&\qquad \leadsto (\mathit{True}, \texttt{'a'}, (2, \mathit{False}, [\mathit{True}, \mathit{False}])) \\
&\mathit{everywhere}\ (\mathit{mkT}\ (\neg))\ (\mathit{True}, (1, \mathit{False}, [\mathit{True}, \mathit{False}])) \\
&\qquad \leadsto (\mathit{False}, (1, \mathit{True}, [\mathit{False}, \mathit{True}]))
\end{aligned}
$$

SYB v současném Haskellu

Současná implementace v GHC:

class *Typeable a* **where**
  typeRep # ::*TypeRep a*

class *Typeable a* ⇒ *Data a* **where**
  gfoldl :: (∀*d b*.*Data d* ⇒ *c* (*d* → *b*) → *d* → *c b*) → (∀*g*.*g* → *c g*) → *a* → *c a*
  gunfold, *toConstr*, *gmapT*, *gmapQ*, *gmapM*, ...

Obě třídy jdou odvodit mechanicky! *Data* navíc zvládá monády a změny typů:

mkT :: (*Typeable a*, *Typeable b*) ⇒ (*b* → *b*) → *a* → *a*
mkQ :: (*Typeable a*, *Typeable b*) ⇒ *r* → (*b* → *r*) → *a* → *r*
mkM :: (*Monad m*, *Typeable a*, *Typeable b*) ⇒ (*b* → *m b*) → *a* → *m a*
mkR :: (*MonadPlus m*, *Typeable a*, *Typeable b*) ⇒ *m b* → *m a*

Generics a okolí

Tradiční jazyky

- *Typeable* zhruba odpovídá RTTI,
  Data velmi zhruba odpovídá něčemu jako “Serializable”
- ‘visitor pattern’ míří podobným směrem, ale neřeší problém $\mathcal{O}(h \cdot v)$ a typicky nejde bezpečně odvodit automaticky.

GHC.Generics Automaticky odvozená typová třída *Generic* poskytuje dost rozsáhlou informaci o typu v GHC, včetně např. jmen konstruktorů.

```
class Generic a where
  type Rep a :: Type → Type
  from :: a → (Rep a) x
  to   :: (Rep a) x → a
```

Uniplate/Biplate Starší a jednodušší reprezentace založená na *from*&*to*; odvoditelná z *Data*.

GHC.Generics

```
ghci> from (Just 3)
M1 {unM1 = R1 (M1 {unM1 = M1 {unM1 = K1 {unK1 = 3}}})}

ghci> :k! Rep (Maybe Int)
Rep (Maybe Int) :: * -> *
= D1
    ('MetaData "Maybe" "GHC.Maybe" "base" 'False)
    (C1 ('MetaCons "Nothing" 'PrefixI 'False) U1
     :+: C1
           ('MetaCons "Just" 'PrefixI 'False)
           (S1
              ('MetaSel 'Nothing
                 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy)
              (Rec0 Int)))
```

Bonus: bez-boilerplatová optika

```
import Control.Lens.Plated

incrVals = transformOf _Val (+1) :: Expr → Expr
incrVals (Plus (Val 1) (Mul (Val 2) (Val 3)))
    ⇝ (Plus (Val 2) (Mul (Val 3) (Val 4)))

(Plus (Val 1) (Var "x")) ^.. cosmos . _Var
    ⇝ ["x"]

import Data.Aeson.Lens
"[1, \"x\", {\"y\": 123}]" ^.. cosmos . _Number
    ⇝ [1,123]
```

Konec!

…ale co dál?

Co dál?

- GADTs, existenciální typy (*GenericList*)
- DataKinds (*API* `"items"`), závislé typy (*Array* `5` *Int*)
- rekurzivní schémata
- efektové systémy (`freer`, `polysemy`, `fused-effects`, `eff`)
- profunktory a profunktorová optika
- šipky (arrows)
- streamové zpracování dat (`conduit`, `pipes`, …)
- lineární typy