Dátové typy

Doplnkový študijný materiál k predmetu IB015

Tomáš Szaniszlo
xszanisz@fi.muni.cz

June 19, 2011

1 Úvod

Dátove typy umožňujú vytvárať nové typy, presnejšie typové konštruktory, z už existujúcich typov (typových konštruktorov). Jednou z ich najdôležitejších vlastností je možnosť vytvárať nové celistvé hodnoty ako kombinácie viacerých hodnôt. Motivačným príkladom môže byť dátový typ zjednodušene popisujúci vypožičateľné jednotky v knižnici:

```
data Unit  =  Book Int String String
              -- kniha: signatura, autor, nazov
           |  Magaz Int String Int
              -- casopis: signatura, nazov, cislo
           |  CD Int String
              -- CD: signatura, popis
```

Takto môžeme mať pre každého používateľa hodnotu `borrowed :: [Unit]` obsahujúcu zoznam vypožičaných vecí, napríklad

```
borrowed  =  [Book 675 "Hemingway" "Starec a more",
              Book 981 "Bulgakov" "Psie srdce",
              CD 45 "Vuvuzela hits"]
```

Takto deklarovaný dátový typ teda umožňuje mať ako hodnoty rovnakého typu (`Unit`) knihy, periodika, aj CD, pričom u každej položky je možné uložiť pre ňu špecifické údaje.

Deklarácia dátového typu vyzerá vo všeobecnosti nasledovne:

```
data TyCon tyvar_1 ... tyvar_n  =  DCon_1 t_11 ... t_1k(1)
                                |  DCon_2 t_21 ... t_2k(2)
                                   ...
                                |  DCon_m t_m1 ... t_mk(m)
  deriving (TyClass_1, ..., TyClass_v)
```

(Platí $n \geq 0, m \geq 1, k(i) \geq 0, v \geq 1$.) Jednotlivé časti deklarácie sú podrobne popísané v ďalších sekciách tohto materiálu.

2 Typové konštruktory

Názov novovytváraného typu je prvou časťou, ktorá nasleduje po uvedení `data`. Vo všeobecnej definícii je označený ako `TyCon`, teda *typový konštruktor*. Typový konštruktor musí vždy začínať veľkým písmenom (ako aj všetky ostatné typy). U jednoduchších typových konštruktorov nasleduje hneď `=` bez nejakých ďalších argumentov ($n = 0$). Takýmito sú napríklad štandardné typové konštruktory `Int`, `Bool`, `Char` a nazývame označujeme ich ako *nulárne*, keďže nevyžadujú žiadny typový argument. Pri vytváraní typov použiť priamo, keďže reprezentujú už konkrétny typ.

3 Argumenty a arita typových konštruktorov

Okrem nulárnych typových konštruktorov sú aj typové konštruktory, ktoré potrebujú nejaké typové argumenty. Spomedzi štandardných sú to `Maybe`, `Either` alebo často definujeme `BinTree` a ich definície vyzerajú takto:

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

Uvedené argumenty `a`, `b` predstavujú *typové premenné* ($\texttt{tyvar}_i$). Typové premenné musia začínať malým písmenom a zvyčajne sú iba jednopísmenové a používajú sa písmena zo začiatku abecedy. V definícii nie je na ich mieste možné použiť konkrétne typy (napr. `Int`, `Double`). Konkrétne typy sa namiesto typových premenných píšu až pri konkrétne vytvorených typoch.

Takto vytvorené typové konštruktory pripomínajú funkcie nad typmi. Ako argumenty im dáme nejaké typy, tieto typy nahradia zodpovedajúce typové premenné na pravej strane `=`, a teda špecializáciou vznikne nejaký konkrétny typ. Môžeme si to predstaviť, ako keby pre každý konkrétny typ, ktorý môžeme dosadiť za typovú premennú, bol definovaný osobitný dátový typ:

```
data Maybe_Int = Nothing | Just Int
data Maybe_(Bool,String) = Nothing | Just (Bool, String)
```

Z tohto hľadiska je to analógiou zvyčajných funkcií na hodnotách, avšak namiesto hodnôt sa pracuje s typmi a dalo by sa písať (hoci takýto zápis sa nepoužíva):

```
TypovyKonstruktor Typ1 Typ2 ⇝ NejakyTyp
```

Je tu však rozdiel v tom, že zatiaľčo u funkcií na hodnotách môžeme používať čiastočnú aplikáciu, teda `div 3` je korektný výraz, u typových konštruktorov to nie je možné. Pri vytváraní typu musíme typový konštruktor aplikovať na presne toľko typov, koľko má v deklarácii typových premenných, ak ho chceme použiť ako súčasť nejakého typu.

Celkovo môžeme typové konštruktory klasifikovať podľa počtu typových premenných, ktoré majú uvedené v deklarácii dátového typu: hovoríme

o nulárnych, *unárnych* (1), *binárnych* (2), *ternárnych* (3) alebo vo všeobecnosti *n-árnych* (n) typových konštruktoroch. Počet typových premenných nazývame aj *arita typového konštruktora* a v na začiatku uvedenej definícii to je n.

4 Dátové konštruktory

Ďalšou časťou deklarácie dátového typu sú *dátové konštruktory* ($\texttt{DCon}_j$). Dátové konštruktory umožňujú identifikovať, ku ktorému typovému konštruktoru (a tým pádom aj typu) daná hodnota patrí a umožňujú združovať v rámci jedného typu hodnoty rôzneho charakteru. Každý typový konštruktor musí mať definovaný aspoň jeden dátový konštruktor ($m \geq 1$).[1] Podobne ako u typových konštruktorov tu platí obmezdenie, že dátové konštruktory musia začínať veľkým písmenom. A ako naznačuje všeobecná definícia, dátový konštruktor sa vyskytuje v jednej časti

Identifikačná úloha spočíva v tom, že ak by sme chceli vytvoriť napríklad dátový typ reprezentujúci hodnoty náhodne generované pomocou mince a hracej kocky,

```
data RandomValue = Coin Bool | Dice Int
data RandomValueBad = Bool | Int
```

ale nepoužili by sme dátové konštruktory, mohli by sme mať hodnotu $True$. Tu by nastal problém s určením jej typu. Štandardne to je hodnota typu `Bool`, ale zároveň by to mohla byť hodnota typu `RandomValueBad`. To znamená, že bez nejakého kontextu by nebolo možné určiť typ výrazu z dvoch možností, ktoré sa navzájom vylučujú, keďže v Haskelli musí mať každá hodnota jediný typ. Avšak ak hodnota začína nejakým dátovým konštruktorom, ten sa môže nachádzať iba v deklarácii jedného typu (nemôžeme ho súčasne použiť na označenie hodnôt v deklarácii iného typu) a takisto v rámci deklarácie jedného typu sa môže vyskytnúť len raz (t.j. $i \neq j \Rightarrow \texttt{DCon}_i \neq \texttt{DCon}_j$). Preto je vďaka dátovým konštruktorom možné jednoznačne priradiť typ takýmto hodnotám: `Coin True :: RandomValue, True :: Bool`.

Jednotlivé dátové konštruktory spolu s ich argumentami od seba oddeľujeme znakom `|`. Tento znak slúži iba na oddelenie jednotlivých zápisov dátových konštruktorov a v typoch alebo hodnotách sa už nevyskytuje.

V súvislosti s dátovými konštruktormi je ešte nutné poznamenať, že sú už konečnou formou hodnôt, teda nie je možné ich ďalej vyhodnotiť tak, ako vyhodnocujeme funkcie. Odtiaľ vlastne pochádza aj ich názov.

[1] Rozšírenie štandardu `EmptyDataDecls` však umožňuje deklaráciu dátových typov bez dátových konštruktorov, avšak takéto typové konštruktory sa už nepoužívajú priamo, keďže obsahujú iba nedefinovanú hodnotu $\perp$. Takéto typové konštruktory spolu s ďalšími prostriedkami jazyka je však možné použiť na dosiahnutie silnejšej typovej kontroly.

5 Argumenty a arita dátových konštruktorov

Podobne ako u typových konštruktorov, aj u dátových konštruktorov môžeme uviesť nejaké argumenty ($\texttt{t}_{ij}$), a podľa ich počtu sa analogickým spôsobom určuje *arita dátového konštruktora*. V definícii na začiatku je arita dátového konštruktora $\texttt{DCon}_i$ rovná $k(i)$ a platí $k(i) \geq 0$.

Pri určovaní arity dátového konštruktora sa zohľadňuje výhradne len počet argumentov bez ohľadu na ich vnútornú štruktúru, teda napríklad v

```
data Student = Stud String Int [Studies]
```

má dátový konštruktor `Stud` aritu 3 bez ohľadu na počet prvkov zoznamu v konkrétnych hodnotách. Arita dátového konštruktora teda nijako nesúvisí s konkrétnymi hodnotami vytvorenými pomocou neho.

Argument $\texttt{t}_{ij}$ určuje typ hodnoty, ktorá sa môže vyskytnúť ako j-ty argument dátového konštruktora $\texttt{DCon}_i$. Tento typ môže byť ľubovoľným konkrétnym typom (teda bez typových premenných), čiže môže byť vytvorený z typových konštruktorov ako `Double`, `Bool`, `(,)`, `[]`, `->`, `BinTree` atď., prípadne môže obsahovať niektoré z typových premenných $\texttt{tyvar}_i$ daného typového konštruktora. Každá z typových premenných, ktoré sa použijú na tvorbu typov $\texttt{t}_{ij}$ musí byť jednou z $\texttt{tyvar}_i$, použitie inej by znamenalo syntaktickú chybu. Zároveň je potrebné použiť v deklarácii daného typu každú typovú premennú aspoň raz.[2] Hoci to vyplýva z definície, je vhodné zdôrazniť, že v argumentoch sa nemôžu nachádzať dátové konštruktory, keďže sú entitami na úrovni hodnôt, a tie sa v typoch nemôžu vyskytovať.

Jedným rozdielom medzi dátovým a typovým konštruktorom je, že zatiaľčo typový konštruktor musí byť aplikovaný vždy na taký počet typov, ako je jeho arita, dátový konštruktor arity m môže byť aplikovaný na k argumentov ($0 \leq k \leq m$) a dostaneme korektnú hodnotu.[3]

Prípad, keď každý dátový konštruktor je nulárny, sa zvykne označovať ako *výpočtový dátový typ* a používa v prípade, keď nejaká premenná môže nadobúdať niekoľko konkrétnych hodnôt:

```
data Colour = Blue | Red | Green | Yellow | Black
```

6 Otypovanie dátových konštruktorov

Dátový konštruktor funguje podobne ako funkcie. Je možné aplikovať ho na hodnoty a ako výsledok dostaneme novú hodnotu. To znamená, že ho môžeme aj otypovať. Majme teda deklaráciu dátového typu

```
data TyCon = DCon t1 t2 ... tk
```

[2] V skutočnosti je možné toto obmedzenie porušiť a v takom prípade dostaneme *fantómový typ*, ktorý je možné výhodne použiť, podobne ako `EmptyDataDecls`, na zosilnenie typového systému.

[3] Nie je však už možné aplikovať viac ako m argumentov čo je možné u niektorých funkcií, napríklad u unárnej funkcie `id :: a -> a` takto: `id id 2` $\rightsquigarrow$ `4`.

a nech platí `x1 :: t1, x2 :: t2, ..., xk :: tk`. Potom

```
DCon x1 x2 ... xk :: TyCon
```

To znamená, že aplikovaním toľkých argumentov (zodpovedajúcich typov), ako je arita dátového konštruktora, dostaneme hodnotu typu `TyCon`. Avšak ako už bolo spomenuté, nie je nutné aplikovať počet argumentov rovný arite. Ak aplikujeme iba prvých i argumentov, prípadne žiadne, dostávame

```
DCon x1 x2 ... xi :: t(i+1) -> ... -> t(k-1) -> tk -> TyCon
DCon :: t1 -> t2 -> ... -> t(k-1) -> tk -> TyCon
```

pričom posledné otypovanie je vlastne *typ dátového konštruktora*.
V konkrétnych prípadoch máme

```
CD 45 "Vuvuzela hits" :: Unit
Book :: Int -> String -> String -> Unit
Just :: a -> Maybe a
Node 'x' ::  BinTree Char -> BinTree Char -> BinTree Char
```

Teraz je dobrá chvíľa spraviť si pauzu pri čítaní, pustiť si obľúbenú hudbu, prejsť sa alebo čokoľvek, a potom sa pustiť do zvyšku tohto dokumentu.

7 Menné priestory konštruktorov

Ako bolo spomenuté v predchádzajúcich častiach, dátové a typové konštruktory musia začínať veľkým písmenom. Môže sa však stať, že definujeme typ, kde bude dátový a typový konštruktor reprezentovaný rovnakým názvom:

```
data Price = Price Int
```

Takáto definícia je korektná a dokonca sa častokrát používa. To je možné vďaka tomu, že menné priestory dátových a typových konštruktorov sú oddelené. V praxi to znamená, že pomenovať dátové a typové konštruktory je možné úplne nezávisle a prekrývanie názvov nerobí žiadny problém, hoci niekedy to môže znižovať prehľadnosť.

Ak máme totiž korektne utvorený výraz alebo program, nikdy nie je možné nahradiť dátový konštruktor typovým konštruktorom a naopak, pretože by vznikol nekorektný výraz. To je dané tým, že dátový konštruktor "pracuje" na úrovni hodnôt, zatiaľčo typový na úrovni typov, a teda vždy pracujeme buď len s hodnotami alebo len s typmi.

8 Rekurzívne typy

Zvláštnym prípadom typov sú *rekurzívne typy*. To znamená, že pri definícii typového konštruktora sa v niektorom argumente $\mathtt{t}_{ij}$ používa práve definovaný typový konštruktor `TyCon`. Takýmito prípadmi sú napriklad typ reprezentujúci prirodzené čísla alebo typ reprezentujúci binárne koreňové stromy s hodnotami v uzloch:

```
data Nat = Zero | Succ Nat
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

Takáto definícia umožňuje zanoriť dátový konštruktor ľubovoľne veľakrát: `Succ (Succ (Succ Zero))`, až nakoniec v istom momente použijeme vetvu deklarácie s nerekurzívnym dátovým konštruktorom.

Dokonca je možné definovať aj typový konštruktor, ktorý bude obsahovať jediný dátový konštruktor, v ktorého argumente bude práve definovaný typový konštruktor:

```
data Rec = Val Rec
```

Typ `Rec` bude mať jedinú plne definovanú hodnotu, ktorá bude obsahovať nekonečne veľa konštruktorov `Val`,[4], aj keď práve takýto definovaný dátový typ nie je veľmi prakticky použiteľný. Jeho existencia a práca s ním je umožnená lenivosťou jazyka. K "nahliadnutiu" dovnútra dátových konštruktorov dôjde len keď to bude nutné.

Podobné hodnoty samozrejme existujú v každom rekurzívnom dátovom type.

Ďalej môžeme definovať nepriamo rekurzívne typové konštruktory:

```
data T1 = Val1 Int T2
data T2 = Val2 Bool T1 | End
```

Typový konštruktor `T1` nie je priamo rekurzívny, ale je možné definovať výraz typu `T1`, ktorý bude obsahovať vlastný podvýraz takisto typu `T1`.

9 Infixové a špeciálne konštruktory

Okrem na začiatku uvedeného obmedzenia na názvy dátovych konštruktorov (musia začínať malým písmenom) je ešte možné vytvoriť aj *infixové dátové konštruktory*, avšak tieto môžu mať len na aritu 2. Infixový dátový konštruktor musí začínať dvojbodkou, po ktorej môžu nasledovať nealfanumerické znaky, napríklad `!`, `*`, `+`, `<`, `>`, `=`, `:` atď., aj keď nie je možné použiť úplne každý.[5] Infixové dátové konštruktory zapisujeme očakávateľným spôsobom:

```
data Infix = Infix :<: Infix | Infix :>: Infix | Val Int
```

a hodnoty takéhoto typu môžu byť napríklad

```
Val 10 :<: Val 3
(:<:)  (Val 0) (Val 1) :>: Val 2
Val 100 :<: (Val 200 :>: Val 0)
```

Ako posledný prípad existujú špeciálne dátové a typové konštruktory, ktoré sú priamo vstavané do syntaxe jayzka Haskell. Týchto niekoľko konštruktorov nepodlieha doteraz spomenutým pravidlám tvorby konštruktorov, majú výsadné postavenie a špeciálnu syntax. Sú to konkrétne (aj s aritou v zátvorke):

[4] Bude obsahovať aj neúplne definované hodnoty $\perp$, `Val` $\perp$, `Val (Val` $\perp$`)`, ...

[5] Presný zoznam sa nachádza v časti 10.2 Lexical Syntax štandardu Haskell 2010 Language Report ako neterminál `symbol`.

Typové konštruktory	
`(->)`	funkcia (2)
`(,)`	usporiadaná dvojica (2)
`(,,)`	usporiadaná trojica (3)
$\underbrace{\texttt{(,...,)}}_{k}$	usporiadaná k-tica (k)
	...
`[]`	zoznam (1)
`()`	nultica (0)

Dátové konštruktory	
`(,)`	usporiadaná dvojica (2)
`(,,)`	usporiadaná trojica (3)
$\underbrace{\texttt{(,...,)}}_{k}$	usporiadaná k-tica (k)
	...
`[]`	zoznam (1)
`(:)`	zoznam (2)
`()`	nultica (1)
`0, 143, -5.4`	čísla (0)
`'a', '*'`	znaky (0)
`"abc", ""`	reťazce (0)

Za zmienku stojí napríklad veľmi známy infixový typový konštruktor `(->)`, ktorý sa používa na vytváranie typov funkcií. Hoci je možné použiť označenie `(->) Int Bool`, takéto sa používa iba v špeciálnych prípadoch; za normálnych okolností sa zapisuje `Int -> Bool`.

Takisto na príklade konštruktorov usporiadaných k-tic vidieť bežnú aplikáciu prekrývania názvov dátových a typových konštruktorov. Podobne ako u funkcií, aj tu sa nezvykne používať prefixový zápis `(,,) Int Bool String`, ale `(Int, Bool, String)`.

Za špeciálny prípad usporiadaných k-tic sa dá považovať nultica, t.j. $k = 0$: `() :: ()`. Je to jediná plne definovaná hodnota daného typu a zároveň spomedzi preddefinovaných nealfanumerických konštruktorov jediný prípad, kde hodnota má rovnakú reprezentáciu, ako jej typ. Zvyčajne sa používa, keď je potrebné zo syntaktických dôvodov použiť nejaký argument, ale sémanticky nie je žiadny potrebný: `putStr :: String -> IO ()` (odoslanie reťazca na štandardný výstup, avšak nie je nejaká rozumná hodnota, ktorú by bolo treba vrátiť), `BinTree ()` (reprezentácia iba štruktúry binárneho stromu neberúc do úvahy hodnoty v uzloch bez nutnosti deklarovať nový dátový typ).

Vzhľadom na infixový zápis konštruktorov, konkrétne `(->)` a `(:)`, je nutné určiť asociativitu, keďže napríklad `a -> (a -> a)`, `(a -> a) -> a` nie sú rovnaké typy. U oboch konštruktorov je to asociativita sprava. U typového konštruktora `(->)` to úzko súvisí so zátvorkovaním pri čiastočnej aplikácii.

Ako posledný, celkom zaujímavý príklad, sú dátové konštruktory reprezentujúce čísla a znaky. Tieto samozrejme nie je možné explicitne definovať, a preto

sú vstavané priamo v jazyku. Čo sa týka reťazcov, tak ich zápis je v skutočnosti len "syntaktickým cukrom". Na nižšej úrovni jazyka bez syntaktického cukru sú rozpísané ako `['a', 'b', 'c']` a `[]`, teda ako zoznamy znakov.

Príklady použitia:

```
Typy                              Hodnoty
Int -> Int                        (curry, id)
Int -> a -> (a -> a) -> a         ((1, 6.5, 'X'), 4)
(Bool, Char) -> (Bool, Bool)      [5,6,7] ≡ 5:6:7:[]
[(Int, [Float])]                  Just ()
String -> ()
```

10 Klauzula `deriving`

Poslednou, voliteľnou časťou deklarácie dátového typu, je klauzula `deriving`. Táto klauzula umožňuje automaticky generovať kód potrebný na to, aby sa stal dátový typ inštanciou požadovanej typovej triedy. Štandardným a explicitným postupom býva

```
instance TyClass TyCon where
  f1 = ...
  f2 = ...
```

Klauzula umožňuje nechať generovanie tohto kódu kompilátoru/interpretu, pokiaľ je to možné. V prípade, že uvádzame len jednu typovú triedu, nie je nutné dávať ju do zátvoriek.

Avšak tento spôsob nie je možné použiť vždy — napríklad nie je možné definovať

```
data FType = F ([Int] -> [Int]) deriving Eq
```

keďže problém určenia rovnosti dvoch všeobecných funkcií nie je algoritmicky riešiteľný.[6]

Medzi typové triedy, ktoré je možné použiť v časti `deriving`, patria `Eq, Ord, Show, Read` a niektoré ďalšie.

11 Používanie dátových typov – vzory

Používanie dátových typov má svoje isté špecifiká. V klasickej definícii funkcie máme napríklad

```
f x = something x (x `something2` 3)
```

teda s vnútornou štruktúrou hodnoty `x` nepracujeme, pracujeme s ňou iba ako s celkom, a tak sa aj odovzdáva ako argument prípadným ďalším funkciám.

[6] Tento problém je pre funkcie s definičným oborom nekonečnej kardinality iba čiastočne rozhodnuteľný.

Pri používaní dátových typov by však nebolo možné rozlíšiť napríklad hodnoty dátového typu `Nat`, keďže definíciou dátového typu nám interpret/kompilátor nedefinuje diskriminátory alebo selektory[7], iba konštruktory.

V Haskelli je možné definovať funkcie, ktoré fungujú ako selektory, avšak častejšie sa využíva prístup *definície podľa vzoru*, ktorá v sebe kombinuje selektory a diskriminátory. Namiesto zapísania formálneho argumentu v hlavičke funkcie sa použije konkrétny dátový konštruktor s formálnymi argumentami, ktorými sa sprístupnia argumenty dátového konštruktora:

```
f :: BinTree a -> Int
f Empty = 0
f (Node val left right) = 1 + f left + f right
```

Jednotlivé definície v uvedenom príklad sa použijú iba pre hodnoty vytvorené zodpovedajúcimi konštruktormi a v druhom prípade sa argumenty sprístupnia v tele funkcie ako `val` (hodnota v uzle), `left` (ľavý podstrom), `right` (pravý podstrom). Dôležité je uvádzať celú konštrukciu v zátvorkách v prípade, že za dátovým konštruktorom nasledujú argumenty, inak by napríklad `fun Node val left right = ...` bolo interpretované ako funkcia, ktorá berie štyri argumenty. Zároveň je nutné zapísať za dátový konštruktor presne toľko argumentov, aká je jeho arita, t.j. nasledovná definícia nie je korektná: `f (Node val) = val`.

Môže sa stať, že chceme definovať už spomínaný selektor, čiže napríklad z binárneho stromu by sme chceli získať hodnotu v uzle. Definícia by mohla vyzerať nasledovne:

```
btVal (Node val left right) = val
```

Avšak zvyšné dva argumenty `left`, `right` vôbec nevyužívame, no napriek tomu ich nejako pomenovávame. V takýchto prípadoch je možné namiesto nepoužitých formálnych argumentov písať podtržítko:

```
btVal (Node val _ _) = val
```

Podtržítko plní úlohu formálneho argumentu, avšak nie je možné odvolávať sa na jeho hodnotu v tele funkcie a zároveň, ako už z príkladu vidno, je možné použiť ho viackrát a vždy predstavuje novú premennú (narozdiel od `btVal (Node val x x) = val`), čo je chybný zápis.

Na záver, vzory môžme používať aj vnorene:

```
g :: BinTree a -> Bool
g (Node _ (Node _ Empty _) _) = True
g _ = False
```

[7] Diskriminátory umožňujú rozlíšiť, ktorým dátovým konštruktorom bola hodnota vytvorená, a selektory umožňujú získať jednotlivé argumenty hodnôt vytvorených pomocou dátových konštruktorov.

12 Zhrnutie

V predchádzajúcich sekciách boli predstavené dátové typy a s nimi súvisiace veci — dátové a typové konštruktory, arita, rekurzívne dátové typy, vzory. Dátové typy sú často používanou konštrukciou jazyka Haskell a prinášajú spomínané výhody celistvosti, jednoznačnej interpretácie, či možnosť kombinácie hodnôt.

Samozrejme majú aj nevýhody, napríklad neobratnosť zápisu v prípade veľkého množstva argumentov dátového konštruktora. Ten sa prejavuje aj u zápisu funkcie, ktorá modifikuje iba jeden argument:

```
data X = A Int String Bool [String] Int (Float, Float) | B

next (A a b c d e f) = A a b c d (succ e) f
next B = B
```

a ešte zdĺhavejší by bol zápis všetkých selektorov na danom dátovom type. Tieto problémy odstraňujú *záznamové dátové typy*.

13 Pokročilejšie a súvisiace témy

Medzi rozšírenia dátových typov alebo s nimi súvisiace témy patria:

- **typové funkcie (synonymá)** - Pomocou kľúčového slova `type` umožňujú definovanie funkcií na typoch. Typickým príkladom je `type String = [Char]`.
- **premenované (obalové) dátové typy** - Umožňujú zaviesť typ hodnôt tak, že obsahovo sú hodnoty identické (izomorfné) s nosným typom, ale označujeme a otypovávame ich inak, teda nie je možné zamieňať ich s hodnotami typu, ktorý obsahuju. Definujeme ich pomocou kľúčového slova `newtype`: `newtype MyInt = IntVal Int`.
- **fantómové typy** - Používanie typových premenných `tyvar`, ktoré sa však nevyskytujú nikde v argumentoch dátových konštruktorov. Umožňujú ďalej jemnejšie deliť typy a často sa využívajú napríklad pri reprezentácii výrazov a ich delenie na číselné výrazy, Boolovské výrazy atď.
- **algebraické dátové typy (ADT)** - Pohľad na dátové typy z hľadiska typovej teórie: súčtové, súčinové typy, rekurzia, atď.
- **generalizované algebraické dátové typy (GADT)** - Zovšeobecnenie ADT. Pre nenulárne typové konštruktory umožňujú špecificky určiť, hodnoty akého typu vracajú jednotlivé dátové konštruktory (napríklad pre `BinTree a` môžeme mať jeden konštruktor, ktorý bude vždy vracať hodnotu typu `BinTree Bool`, zatiaľčo ostatné budú vracať hodnoty typu `BinTree a`.

- **typové triedy** - Umožňujú preťažovať funkcie tak, aby boli definované na viacerých typoch. V istom zmysle predstavujú "supertypy" — zoskupenia typov, u ktorých je možné používať spoločné funkcie. Spomedzi známejších sú to napríklad `Eq`: `(==)`, `(/=)` alebo `Monad`: `(>>=)`, `(>>)`, `return`, `fail`. Ich zovšeobecnením sú viacparametrické typové triedy (multiparameter type classes, MPTC).
- **druhy (kinds)** - Sú analógiou typov. Tak, ako typy klasifikujú hodnoty, kindy klasifikujú typy (typové konštruktory). Niekedy sa využívajú pri definovaní typových tried.
- **striktné dátové typy** - Dátové typy s argumentami, ktoré sa vyhodnotia okamžite (striktne, nie lenivo), keď sa vytvára hodnota.

14 Ďalšie zdroje informácií

- `http://www.haskell.org/haskellwiki/Type`
- `http://en.wikibooks.org/wiki/Haskell/Type_declarations`
- `http://naucte-se.haskell.cz/vytvarime-si-sve-typy-a-typove-tridy`
- `http://www.fi.muni.cz/~xnovak34/haskellhero/index.php?page=lessons&lesson=62`
- `http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-680004.2`