Funkcionální programování

Algebraické datové typy (ADT)

Petr Pudlák

21. prosince 2011

Osnova

Skládání funkcí

Cvičení

Definujte funkci skládání funkcí $\circ$ jako lambda term.
Určete její typ.

Osnova

Direktní součty a součiny

V programech potřebujeme mít možnost skládat složitější datové struktury z jednodušších.
Matematicky můžeme rozlišit 2 základní operace:

- Direktní součin $\otimes$. Výsledná hodnota „zabaluje" několik dílčích hodnot.
- Direktní součet $\oplus$. Výsledná hodnota zabaluje právě jednu z několika možných hodnot. [1] Obvykle se používá na větvení algoritmů podle struktury dat.

[1] V procedurálních jazycích se direktní součet vyskytuje poměrně zřídka.

Direktní součin

Definition ($\otimes$ – kategorická definice)

Direktní součin objektů A_j, $j \in \mathcal{J}$ je objekt[a] A spolu se zobrazeními[b] (morfismy) $p_j : A \to A_j$ takovými, že pro libovolný objekt B a zobrazení $f_j : B \to A_j$ existuje zobrazení $f : B \to A$ takové, že $f_j = p_j \circ f$ pro všechna $j \in \mathcal{J}$.
Zobrazení p_j se nazývají *kanonické projekce*.
Značení: $\bigotimes$ či $\prod$. Funkci f značíme někdy $\bigotimes_{\{j \in \mathcal{J}\}} f_j$.

---

[a] V našem případě jsou objekty (tedy domény a kodomény funkcí) typy.
[b] V našem případě jsou zobrazení lambda funkce.

(Obrázek.)

Direktní součet

Definition ($\oplus$ – kategorická definice)

Direktní součet objektů A_j, $j \in \mathcal{J}$ je objekt A spolu se zobrazeními (morfismy) $i_j : A_j \to A$ takovými, že pro libovolný objekt B a zobrazení $f_j : A_j \to B$ existuje zobrazení $f : A \to B$ takové, že $f_j = f \circ i_j$ pro všechna $j \in \mathcal{J}$.
Zobrazení i_j se nazývají *kanonická vnoření*.
Značení: $\bigoplus$ či $\coprod$. Funkci f značíme někdy $\bigoplus_{\{j \in \mathcal{J}\}} f_j$.

(Obrázek.)

Direktní součty a součiny v lambda kalkulu

V typovaném i netypovaném lambda kalkulu můžeme zapsat direktní součty i součiny pomocí lambda termů.

Cvičení

Jak? Navrhněte lambda termy, které implementují tyto funkce, a zapište jejich typy.

Kontinuace

Funkce typu

$$(\zeta \to \alpha) \to \alpha$$

se často nazývají *kontinuace*.
Typ takové funkce říká: „Mám hodnotu typu ζ. Když mi řekneš, kam ji dosadit, vypočtu Ti výsledek.”

Direktní součin v lambda kalkulu

Mějme typy τ_1 a τ_2 definujme jejich direktní součin jako kontinuaci:

$$\tau = \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma$$

Direktní součin v lambda kalkulu

Mějme typy τ_1 a τ_2 definujme jejich direktní součin jako kontinuaci:

$$\tau = \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma$$

Pro $f_1 : \rho \to \tau_1$ a $f_2 : \rho \to \tau_2$ zaveďme

$$(f_1 \otimes f_2) = \lambda x.\lambda f.f(f_1 x)(f_2 x) \qquad (f_1 \otimes f_2) : \rho \to \tau$$

Kanonické projekce pak vyjádříme:

$$p_1 = \lambda c.c(\lambda xy.x) \qquad p_1 : \tau \to \tau_1$$
$$p_2 = \lambda c.c(\lambda xy.y) \qquad p_2 : \tau \to \tau_2$$

Direktní součet v lambda kalkulu

Mějme typy τ_1 a τ_2 definujme jejich direktní součet jako kontinuaci:

$$\tau = \forall\gamma.(\tau_1 \to \gamma) \to (\tau_2 \to \gamma) \to \gamma$$

Direktní součet v lambda kalkulu

Mějme typy τ_1 a τ_2 definujme jejich direktní součet jako kontinuaci:

$$\tau = \forall\gamma.(\tau_1 \to \gamma) \to (\tau_2 \to \gamma) \to \gamma$$

Kanonická vnoření definujme:

$$i_1 = \lambda x.(\lambda gh.gx) \qquad i_1 : \tau_1 \to \tau$$
$$i_2 = \lambda y.(\lambda gh.hy) \qquad i_2 : \tau_2 \to \tau$$

a pro $f_1 : \tau_1 \to \rho$ a $f_2 : \tau_2 \to \rho$ zaveďme

$$(f_1 \oplus f_2) = \lambda c.cf_1 f_2 \qquad (f_1 \oplus f_2) : \tau \to \rho$$

Kombinace $\otimes$ a $\oplus$

V praxi se ukazuje jako užitečné používat tuto kombinaci:

$$(\tau_{11} \otimes \ldots \otimes \tau_{1k_1}) \oplus \ldots \oplus (\tau_{n1} \otimes \ldots \otimes \tau_{nk_n})$$

(připomíná disjunktivní normální formu).
Její kontinuace pak je tvaru:

$$\forall\gamma.(\tau_{11} \to \ldots \to \tau_{1k_1} \to \gamma) \to \ldots \to (\tau_{n1} \to \ldots \to \tau_{nk_n} \to \gamma) \to \gamma$$

Nevýhody

- Tyto direktní součiny a sumy nejsou nijak pojmenované.
- Chybí možnost rekurzivních typů.
- Naše kontinuace jsou univerzálně kvantifikované, proto je problém přímo s nimi pracovat v Hindley-Milnerově typovém systému.

Osnova

Rozšiřování typového systému o nové symboly

- Náš typový systém obsahuje nyní jen jeden operátor na typech: $\rightarrow$.
- Chceme umožnit přidání nových symbolů do typového systému, které budou reprezentovat a pojmenovávat nové polymorfní typy.
- Potřebujeme umět rozlišit, jak tyto funkce na typech pracují.

Druhy (kinds)

Zavedeme další syntaktický pojem *druh (kind)*, který bude vůči typům to samé, co jsou typy vůči termům.

Definition (druhy)

Druhy definujeme induktivně:

$$\mathbb{K} = * \mid \mathbb{K} \Rightarrow \mathbb{K}$$

- Druh $*$ reprezentuje typy.
- Složené druhy reprezentují operace nad typy.
- V Haskellu se používají stejné šipky pro druhy i typy.
  Pro zpřehlednění budeme pro druhy používat $\Rightarrow$.

Odvození druhů v Haskellu

- Před odvozováním typů se odvozují druhy jednotlivých typových výrazů.
- Odvození druhů je jednodušší v tom, že:
  - druhy neobsahují druhové proměnné a že
  - systém druhů je monomorfní (není polymorfní).
- Základní pravidlo je podobné jako u typů:

$$\text{(eliminace } \Rightarrow) \frac{\alpha : (\kappa_1 \Rightarrow \kappa_2) \qquad \beta : \kappa_1}{(\alpha\beta) : \kappa_2}$$

- Pokud při odvozování druhu vychází nejednoznačnost (místo pro druhovou proměnnou), použije se implicitně druh $*$.
  Příklad: při deklaraci *x* :: *a b* by obecně mohly mít proměnné *a* a *b* druhy $\kappa \Rightarrow *$ a κ; odvozené druhy tedy budou $* \Rightarrow *$ a $*$.

Osnova

Algebraické datové typy – ADT[2]

Cíl: chceme zavést nový typ izomorfní s

$$(\tau_{11} \otimes \ldots \otimes \tau_{1k_1}) \oplus \ldots \oplus (\tau_{n1} \otimes \ldots \otimes \tau_{nk_n})$$

[2] Neplést s *abstraktními datovými typy*!

Algebraické datové typy – ADT[2]

Cíl: chceme zavést nový typ izomorfní s

$$(\tau_{11} \otimes \ldots \otimes \tau_{1k_1}) \oplus \ldots \oplus (\tau_{n1} \otimes \ldots \otimes \tau_{nk_n})$$

Buď $\{\alpha_1, \ldots, \alpha_m\} = \bigcup_{ij} \mathrm{FTV}(\tau_{ij})$ a buď κ_i odvozený druh typové proměnné α_i.
Do jazyka typů přidáme nový symbol (typovou konstantu), který zde označme C, a který reprezentuje nově vytvářený typ.
Druh tohoto symbolu bude $C : \kappa_1 \Rightarrow \ldots \Rightarrow \kappa_m \Rightarrow *$.

[2] Neplést s *abstraktními datovými typy*!

ADT – symboly do jazyka termů

Do jazyka termů přidáme n nových konstant C_i ($i \in \{1, \ldots, n\}$), které nezveme *konstruktory*, a přidáme axiomy:

$$\frac{}{\Gamma \vdash C_i : \tau_{i1} \to \ldots \to \tau_{ik_i} \to C\alpha_1 \ldots \alpha_n}\ ^{3}$$

Dále přidáme do jazyka termů konstantu case_C reprezentující *dekonstrukční funkci*:

$$\frac{}{\begin{array}{ll}\Gamma \vdash \mathrm{case}_C : C\alpha_1 \ldots \alpha_n & \to (\tau_{11} \to \ldots \to \tau_{1k_1} \to \gamma) \to \quad \ldots \\ & \to (\tau_{n1} \to \ldots \to \tau_{nk_n} \to \gamma) \to \quad \gamma\end{array}}$$

Poznámka: Ve funkcionálních jazycích se obvykle dekonstrukční funkce explicitně nevyskytují. Místo nich se používá uživatelsky přátelštější *rozpoznávání vzorů (pattern matching)*, viz. následující kapitola.

[3] Type checker musí odvodit povolené druhy pro α_i.

ADT – redukční pravidla

Konečně přidáme nová redukční pravidla:

$$\text{case}_C\ (C_i\, e_1 \ldots e_{k_i})\, f_1 \ldots f_n \ \rightarrow_{C_i}\ f_i\, e_1 \ldots e_{k_i}$$

Rekurzivně definované typy

- Jestliže se C vyskytuje uvnitř některého z τ_{ij}, říkáme, že typ C je *rekurzivní*.
- Funkcionální jazyky v podstatě vždy umožňují použití rekurzivních typů (případně s určitými omezeními).
- O různých variantách rekurzivních typů pojednávají následující kapitoly.

Osnova

Datové typy v Haskellu

Datový typ C izomorfní s

$$(\tau_{11} \otimes \ldots \otimes \tau_{1k_1}) \oplus \ldots \oplus (\tau_{n1} \otimes \ldots \otimes \tau_{nk_n})$$

a s konstruktory $C_i : \tau_{i1} \to \ldots \to \tau_{ik_i} \to C\,\alpha_1 \ldots \alpha_n$ v Haskellu zapíšeme:

data $C\ \alpha_1 \ldots \alpha_m = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_n\ \tau_{n1} \ldots \tau_{nk_n}$

Poznámka: Posloupnostmi symbolů začínající na „:” lze definovat infixní konstruktory.

Projekční funkce v Haskellu

- Při definici datových typů můžeme zároveň definovat projekční funkce pro jednotlivé konstruktory.
- Například místo

$$\begin{aligned}\textbf{data}\ \mathit{Tree3}\ a = {} & \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree3}\ a)\ (\mathit{Tree3}\ a) \\ & \mid \mathit{Tri}\ \ (\mathit{Tree3}\ a)\ (\mathit{Tree3}\ a)\ (\mathit{Tree3}\ a)\end{aligned}$$

můžeme psát

$$\begin{aligned}&\textbf{data}\ \mathit{Tree3}\ a = \\ &\quad \mathit{Leaf}\ a \\ &\mid \mathit{Bin}\ \{\mathit{left} :: \mathit{Tree3}\ a, \qquad\qquad\qquad\quad \mathit{right} :: \mathit{Tree3}\ a\} \\ &\mid \mathit{Tri}\ \ \{\mathit{left} :: \mathit{Tree3}\ a, \mathit{middle} :: \mathit{Tree3}\ a, \mathit{right} :: \mathit{Tree3}\ a\}\end{aligned}$$

- Pokud má datový typ právě jeden konstruktor (je tedy čistě direktním součinem), odpovídají projekční funkce kanonickým projekcím z definice direktního součinu.
- Pokud má datový typ více konstruktorů, projekční funkce nemusí být totální. Mohou být ale definovány pro více konstruktorů, jako v našem příkladu.

Konstrukce *type*

- Konstrukce **type** umožňuje definovat (parametrizované) synonymum pro typ.
- Původní typ lze libovolně zaměňovat s novým synonymem.
- Nelze použít rekurzivně.
- Příklad:

$$\begin{aligned}
&\textbf{type } \mathit{String} &&= [\mathit{Char}] \\
&\textbf{type } \mathit{Name} &&= \mathit{String} \\
&\textbf{type } \mathit{ListMap}\ a\ b &&= [(a, b)]
\end{aligned}$$

Konstrukce *newtype* v Haskellu I

- Konstrukce **newtype** deklaruje nový typ, jehož vnitřní reprezentace je stejná jako originálního typu.
- Zabalování/rozbalování konstruktoru může být implementováno tak, aby ve výsledném programu nepřinášelo žádné zdržení.
- Musí mít právě jeden konstruktor s právě jedním argumentem.
  Například:
  newtype $Indexed\ a = Indexed\ (a, Int)$
- Může definovat rekurzivní typy.
- Konstruktor deklarovaný pomocí **newtype** je striktní (AKA *unlifted*)!
  To znamená, že $C\ \bot = \bot$.

Konstrukce *newtype* v Haskellu II

newtype se často používá, pokud potřebujeme definovat pro jeden typ různé instance tříd. Například:

```
data Pair a b = a :*: b

instance Functor (Pair a) where
  f `fmap` (a :*: b) = a :*: f b

newtype RevPair a b = RevPair (Pair b a)

instance Functor (RevPair b) where
  f `fmap` (RevPair (a :*: b)) = RevPair (f a :*: b)
```

Příklady I

Example

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

Dekonstrukční funkce by v Haskellu měla typ:

$$\mathit{caseBool} :: \mathit{Bool} \to c \to c \to c$$

Příklady II

Example (Direktní součin)

V Haskellu je speciální syntaktická konstrukce pro páry (a vícenásobné direktní součiny): Z typů a a b sestrojíme pár výrazem (a, b). Stejně zapíšeme i konstruktor pro pár.
Příklad:

$$sqrt' :: Double \to (Double, Double)$$
$$sqrt'\ x = \textbf{let}\ sq = sqrt\ x\ \textbf{in}\ (sq, -sq)$$

Také je možno zapsat typ a konstruktor prefixním zápisem:

$$sqrt' :: Double \to (,)\ Double\ Double$$
$$sqrt'\ x = \textbf{let}\ sq = sqrt\ x\ \textbf{in}\ (,)\ sq\ (-sq)$$

Pro binární direktní součin (pár) jsou v Haskellu definovány projekční funkce *fst* a *snd*.

Příklady III

Example (Jednotkový typ (0-ární direktní součin))

$$\textbf{data}\ \mathit{Unit} = \mathit{Unit}$$

V Haskellu je speciální syntaktická konstrukce pro jednotkové typy. Typ i konstruktor se zapisuje (), tedy jako nulární direktní součin.
Dekonstrukční funkce by měla typ:

$$\mathit{caseUnit} :: () \to c \to c$$

Poznámky:

- V jazyce s líným vyhodnocováním je typ a izomorfní s typem $() \to a$.
- Pro každý typ a existuje právě jedna[a] totální funkce typu $a \to ()$.

[a] vzhledem k extenzionální rovnosti funkcí.

Příklady IV

Example (Direktní součet)

Ve standardní knihovně Haskellu je definován datový typ reprezentující binární direktní součet:

$$\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

Konstruktory *Left* a *Right* odpovídají kanonickým vnořením.
Dekonstrukční funkce by měla typ:

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

Příklady V

Example (0-ární součet)

V Haskellu nelze standardně definovat prázdný typ, který by neměl žádné konstruktory. S rozšířením *EmptyDataDecls* v GHC je to možné:

```
{-# LANGUAGE EmptyDataDecls #-}
data Empty
```

Dekonstrukční funkce by měla typ:

$$caseEmpty :: Empty \rightarrow c$$

(Pokud vím, v současnosti nejde v Haskellu takovou funkci zapsat, viz. GHC #2431.)

Příklady VI

Maybe

- Kontejner obsahující buďto hodnotu nebo označení neúspěchu, je v Haskellu definován takto:

  data *Maybe a* = *Nothing* | *Just a*

Příklady VII

Seznamy

Pro seznamy jsou v Haskellu definovány speciální symboly.
V pseodu-kódu bychom definici Haskellovských seznamů zapsali:

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

nebo (téměř se nepoužívá):

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

- Typy:
  - [] je typ pro datový typ seznamu, můžeme například psát [] *Int*.
  - Téměř vždy se používá zápis [*Int*] apod.
- Termy:
  - [] je konstruktor prázdného seznamu.
  - Symbol : je konstruktorem přidávajícím data na začátek seznamu.
  - Místo $e1 : e2 : \ldots : eN : []$ lze použít zápis $[e1, e2, \ldots, eN]$.

Příklady VII

Seznamy – *List Comprehensions*

- Pro sofistikovanou konstrukci seznamů lze použít syntaktický cukr zvaný *list comprehension*, viz.
  - The Haskell 98 Report,
  - Haskell/List Processing (Wikibooks),
  - List comprehension (Haskell Wiki).
- Místo *list comprehension* lze použít *MonadPlus* [].

Příklady VIII

Example

Přirozená čísla lze podle vzoru Peanovy aritmetiky definovat:

$$\textbf{data}\ \mathit{Nat} = \mathit{Nat} \mid \mathit{Suc}\ \mathit{Nat}$$

Example

Datové typy mohou i ryze nekonečné, jako například:

$$\textbf{data}\ \mathit{Process}\ a = a :> \mathit{Process}\ a$$

(Zde jsme použili infixní konstruktor $:>$.)

Cvičení

- *Jak by vypadaly dekonstrukční funkce těchto typů?*
- *Jak bychom definovali Process ve striktním jazyce?*

Osnova

Striktní konstruktory

Striktní funkce

Označme symbolem $\bot$ program, který nevrátí hodnotu (zacyklí se, případně zhavaruje). Jako funkci můžeme tento symbol definovat například takto:

$$\bot :: a$$
$$\bot = \bot$$

Při popisování sémantiky programů nás často zajímá, jak se chová funkce, když jeden nebo více jejích argumentů je $\bot$.

Definition

Funkci f nazveme *striktní*, pokud $f \bot = \bot$. V opačném případě se nazývá *nestriktní*, někdy též nesprávně *líná*.

Striktní konstruktory

- Haskell umožňuje deklarovat konstruktory tak, aby byly striktní v některých svých argumentech.
- Při deklaraci konstruktoru označíme takový argument tím, že před něj napíšeme symbol !.
- Při aplikaci konstruktoru jsou striktní argumenty vyhodnoceny do WHNF, než jsou dosazeny do konstruktoru:
- Příklad: Seznam striktní v hodnotách. Nemůže obsahovat hodnotnu $\perp$, ale může být nekonečný:

data *StrictList1 a* = *Nil1* | *Cons1* ! *a* (*StrictList1 a*)

- Příklad: Seznam striktní v podseznamech. Může obsahovat hodnotnu $\perp$, ale nemůže být nekonečný:

data *StrictList2 a* = *Nil2* | *Cons2 a* ! (*StrictList1 a*)

Osnova

Osnova

Visitor design pattern v OOP I

- Větvení kódu podle disjunktních dat (direktního součtu) můžeme v OOP implementovat pomocí návrhového vzoru *visitor*.
- Datový typ direktního součtu je deklarován jako abstraktní třída.
- Jednotlivé direktní sčítance jsou reprezentovány jako její podtřídy.
- Kód, který je dosazován do case funkce, je deklarován pomocí interface zvaného *visitor*.

Visitor design pattern v OOP II

```
abstract class Maybe<T> {
    abstract <R> R caseMaybe(MaybeVisitor<T,R> visitor);
}
class Nothing<T> extends Maybe<T> {
    <R> R caseMaybe(MaybeVisitor<T,R> visitor) {
        return visitor.visit(this);
    }
}
class Just<T> extends Maybe<T> {
    T value;

    <R> R caseMaybe(MaybeVisitor<T,R> visitor) {
        return visitor.visit(this);
    }
}

interface MaybeVisitor<T,R> {
    R visit(Nothing<T> data);

    R visit(Just<T> data);
}
```

Visitor design pattern v OOP III

```
class Example {
    void print(Maybe<Integer> data)
    {
        System.out.println(data.caseMaybe(
            new MaybeVisitor<Integer,String>() {
                public String visit(Nothing<Integer> data)
                { return "Nothing"; }

                public String visit(Just<Integer> data)
                { return "Just " + data.value; }
        }));
    }
}
```

Osnova

Vzor

Definition (Vzor)

Vzor (pattern) p typu τ je buďto:

- proměnná, nebo
- $Cp_1 \ldots p_n$, kde C je jeden z konstruktorů typu τ arity n a p_i jsou vzory typů odpovídající typům argumentů konstruktoru C.

Definice funkce vzory

Definition (Definice funkce pomocí vzorů)

Funkci typu $f : \tau_1 \to \ldots \to \tau_n \to \rho$ definujeme (v Haskellu) zápisem

$$
\begin{aligned}
& f :: \tau_1 \to \ldots \to \tau_n \to \rho \\
& f\ p_{11} \ldots p_{1n} \mid g_1 = e_1 \\
& \ldots \\
& f\ p_{k1} \ldots p_{kn} \mid g_k = e_k
\end{aligned}
$$

Kde p_{ij} jsou vzory typů τ_j a $e_i :: \rho$.
Výrazy $g_i :: \mathit{Bool}$ nazýváme *hlídače (guards)*.
Vzor nemusí mít hlídače, pak se $\mid$ vůbec nepíše.
Hlídač může být zapsán bez vzoru (pouze „$\mid g$"), pak kopíruje předchozí vzor.
Ve výrazech g_i a e_i je možno používat proměnné vázané ve vzorech.

Sémantika rozpoznávání vzorů

Neformálně:

- Program probírá vzory v pořadí, v jakém jsou definovány.
- Pokud vstupní data odpovídají vzorům na i-tém řádku, a výraz v hlídači se vyhodnotí na *True*, výsledkem je výraz e_i.
- Pokud vstupní data neodpovídají žádnému vzoru (a hlídači), výsledkem je $\bot$.

Konstrukce **case** v Haskellu

V Haskellu je možné rozpoznáváním vzorů zapsat výraz pomocí konstrukce **case**.
Například:

```
toList :: Either Int a → [Int]
toList val = 3 : case val of
                   Left v  → [v]
                   Right _ → []
```

Znak _ zastupuje nepojmenovanou proměnnou ve vzoru.
Programátor dává čtenáři kódu najevo, že hodnota se nikde nepoužije.

Překlad vzorů na case funkce

Každý vzor se dá přeložit pomocí case... funkcí.
Následující algoritmus ukazuje jednu z možností takového překladu.
(Podrobně popsáno např. v [1, kap. 5].)
Pro potřeby algoritmu si zaveďme pomocné značení

$$\text{case}\left[\begin{array}{ccc|c|c} p_{11} & \cdots & p_{1n} & g_1 & \\ \cdots & & \cdots & \cdots & \longrightarrow \\ p_{k1} & \cdots & p_{kn} & g_k & \\ & & & & \end{array}\begin{array}{c} e_1 \\ \cdots \\ e_k \\ e \end{array}\right]$$

pro anonymní funkci vytvořenou rozpoznáváním vzorů.
Algoritmus zjednodušuje vzory podle prvního sloupce.

Překlad vzorů na case funkce

Zjednodušujme blok

$$\text{case}\left[\begin{array}{ccc|c|c} p_{11} & \cdots & p_{1n} & g_1 & \\ \cdots & & \cdots & \cdots & \longrightarrow \\ p_{k1} & \cdots & p_{kn} & g_k & \\ & & & & \end{array}\begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{c} e_1 \\ \cdots \\ e_k \\ e \end{array}\right]$$

který obsahuje

- k vzorů o n vstupech,
- každý vzor (volitelně) s hlídačem g_j a s výslednou hodnotou e_j, a konečně
- funkci e, které se argumenty předají v případě, že argumenty nerozpozná žádný ze vzorů;
  (na začátku je $e = \mathit{undefined}$).

Překlad vzorů na case funkce

Žádný vzor

Pokud nemá blok pro n vstupů žádný vzor, vyhodnotí se na e.

Překlad vzorů na case funkce

Prázdné vzory

Pokud jsou vzory prázdné, vyhodnotí se na

$$\text{if } g_1 \text{ then } e_1 \text{ else case } \left[\!\!\left[\begin{array}{cc} g_2 & e_2 \\ \dots & \dots \\ g_n & e_n \\ & e \end{array} \right]\!\!\right]$$

Pokud g_1 chybí, nebo je *True*, nahradíme výraz jen e_1 a zbytek vynecháme. (V Haskellu je pro lepší čitelnost hlídačů definována konstanta *otherwise* = *True*.)

Překlad vzorů na case funkce

p_{11} je proměnná

- Je-li p_{11} proměnná, vybereme všech prvních l vzorů, které mají rovněž v prvním sloupci proměnnou.
- Tyto proměnné přejmenujeme na jedinou proměnnou x.
- Potom vzor zjednodušíme na

$$\lambda x\,.\,\text{case}\left[\begin{array}{ccc|c|c} p_{12} & \cdots & p_{1n} & g_1 & e_1 \\ \cdots & & \cdots & \cdots & \cdots \\ p_{l2} & \cdots & p_{ln} & g_l & e_l \\ & & & & e'\,x \end{array}\right]$$

kde

$$e' = \text{case}\left[\begin{array}{ccc|c|c} p_{(l+1)1} & \cdots & p_{(l+1)n} & g_{(l+1)} & e_{(l+1)} \\ \cdots & & \cdots & \cdots & \cdots \\ p_{k1} & \cdots & p_{kn} & g_k & e_k \\ & & & & e \end{array}\right]$$

Překlad vzorů na case funkce I

p_{11} je konstruktor

- Je-li p_{11} konstruktor, vybereme všech prvních l vzorů, které mají rovněž v prvním sloupci jakýkoliv konstruktor (daného typu C).
- V těchto vzorech pak seskupíme stejné konstruktory k sobě (stabilní třídění!), a blok zjednodušíme na

$$\lambda x \,.\, \underset{C}{\mathrm{case}}\, x\, f_1\ \ldots\ f_t$$

kde t je počet konstruktorů daného typu C.
Pro každý konstruktor C_j arity u, pro který jsou dány vzory

$$\begin{array}{llll|ll} C_j(q_{m1}\ldots q_{mu}) & p_{m2} & \cdots & p_{mn} & g_m & e_m \\ \ldots & & & & \ldots & \ldots \\ C_j(q_{m'1}\ldots q_{m'u}) & p_{m'2} & \cdots & p_{m'n} & g_{m'} & e_{m'} \end{array}$$

Překlad vzorů na case funkce II

p_{11} je konstruktor

zavedeme

$$f_j = \text{case} \left[\begin{array}{ccccccc|c} q_{m1} & \cdots & q_{mu} & p_{m2} & \cdots & p_{mn} & g_m & & e_m \\ \cdots & & & & & & \cdots & \longrightarrow & \cdots \\ q_{m'1} & \cdots & q_{m'u} & p_{m'2} & \cdots & p_{m'n} & g_{m'} & & e_{m'} \\ & & & & & & & & e'x \end{array} \right]$$

kde

$$e' = \lambda y_1 \ldots y_u \,.\, \text{case} \left[\begin{array}{ccc|c} p_{(l+1)1} & \cdots & p_{(l+1)n} & g_{(l+1)} & & e_{(l+1)} \\ \cdots & & \cdots & \cdots & \longrightarrow & \cdots \\ p_{k1} & \cdots & p_{kn} & g_k & & e_k \\ & & & & & e \end{array} \right]$$

Příklad I

Spojování seznamů pomocí rozpoznávání vzorů:

$$
\begin{aligned}
&\mathit{merge} :: (a \to a \to \mathit{Bool}) \to [a] \to [a] \to [a] \\
&\mathit{merge}\ \_\ \mathit{xs}\ [\,] = \mathit{xs} \\
&\mathit{merge}\ \_\ [\,]\ \mathit{ys} = \mathit{ys} \\
&\mathit{merge}\ \mathit{cmp}\ (x : \mathit{xs})\ (y : \mathit{ys}) \\
&\quad = \textbf{if}\ x\ `\mathit{cmp}`\ y \\
&\qquad \textbf{then}\ y : \mathit{merge}\ \mathit{cmp}\ (x : \mathit{xs})\ \mathit{ys} \\
&\qquad \textbf{else}\ x\ \ : \mathit{merge}\ \mathit{cmp}\ \mathit{xs}\ \ \ \ \ \ (y : \mathit{ys})
\end{aligned}
$$

Spojování seznamů pomocí *case* funkcí:

$$
\begin{aligned}
&\mathit{caseL} :: [a] \to c \to (a \to [a] \to c) \to c \\
&\mathit{caseL}\ [\,]\ \ \ \ \ \ v\ \_ = v \\
&\mathit{caseL}\ (x : \mathit{xs})\ \_\ f = f\ x\ \mathit{xs}
\end{aligned}
$$

Příklad II

```
merge' :: (a → a → Bool) → [a] → [a] → [a]
merge' = λcmp →
  let
    g = λxs → caseL xs (λys → ys)
      (λx xs' ys → caseL ys ⊥
        (λy ys' → if x `cmp` y
          then y : merge' cmp xs  ys'
          else x  : merge' cmp xs' ys
        )
      )
  in λxs ys → caseL ys xs (\_ _ → g xs ys)
```

Cvičení

Cvičení

(Copied from Chap. 4 of A Gentle Introduction to Haskell, Version 98.)
What's the difference between these two functions?

$$
\begin{aligned}
&take\ 0\ \_ &&= [\,] \\
&take\ \_\ [\,] &&= [\,] \\
&take\ n\ (x : xs) &&= x : take\ (n-1)\ xs
\end{aligned}
$$

And this slightly different version (the first 2 equations have been reversed):

$$
\begin{aligned}
&take1\ \_\ [\,] &&= [\,] \\
&take1\ 0\ \_ &&= [\,] \\
&take1\ n\ (x : xs) &&= x : take1\ (n-1)\ xs
\end{aligned}
$$

Příklad – typové proměnné netriviálního druhu I

Datový typ pro zobecněné stromy, kde je větvení stromu určeno parametrem typové funkce *m*:

```
data TreeList m a = Nil | a :> (m (TreeList m a))
```

Deklarujeme, že konstruktor asociuje doprava a má nižší prioritu:

```
infixr 5 :>
```

Funkce, která počítá počet prvků seznamu (je-li dána převodní funkce z *m* do []):

```
count :: (m (TreeList m a)) → [TreeList m a]) → TreeList m a → Int
count expand = trv
  where
    trv Nil       = 0
    trv (x :> xs) = 1 + sum (map trv (expand xs))
```

Příklad – typové proměnné netriviálního druhu II

Seznam můžeme zadefinovat jako zobecněný strom, který má vždy jednu větev:

$$\textbf{newtype}\ \mathit{Id}\ a = \mathit{Id}\ \{\mathit{unid} :: a\}$$
$$\textbf{type}\ \mathit{List}\ a = \mathit{TreeList}\ \mathit{Id}\ a$$
$$\mathit{countList} = \mathit{count}\ ((:[]) \circ \mathit{unid})$$

Binární strom popíšeme jako zobecněný strom s dvěma větvemi:

$$\textbf{data}\ \mathit{Pair}\ a = \mathit{Pair}\ \{\mathit{pairL} :: a, \mathit{pairR} :: a\}$$
$$\textbf{type}\ \mathit{BinTree}\ a = \mathit{TreeList}\ \mathit{Pair}\ a$$
$$\mathit{countBinTree} = \mathit{count}\ (\lambda p \to [\mathit{pairL}\ p, \mathit{pairR}\ p])$$

Strom s libovolným počtem větví popíšeme pomocí seznamu:

Příklad – typové proměnné netriviálního druhu III

$$\textbf{type}\ \mathit{NTree}\ a = \mathit{TreeList}\ [\,]\ a$$

$$\mathit{countNTree} = \mathit{count}\ \mathit{id}$$

Striktní rozpoznávání vzorů

- Z naší definice rozpoznávání vzorů plyne, že datový typ je vyhodnocen do WHNF před tím, než je na něj aplikováno (aby bylo možno poznat, jakým konstruktorem byl vytvořen).
- Toto nazvěme *striktní rozpoznávání vzorů*.
- Zejména platí, že
  case $\perp$ **of**... $= \perp$

Nestriktní rozpoznávání vzorů I

- V některých případech je výhodné, aby bylo rozpoznávání vzorů nestriktní, tedy aby bylo vyhodnocení termu do WHNF odloženo až do chvíle, kdy je skutečně použit rozpoznaný argument konstruktoru.
- V Haskellu deklarujeme nestriktní vzor tak, že před něj zapíšeme symbol $\sim$.
  - Takový vzor uspěje vždy, proto jej nazýváme *neodmítnutelný*.
  - Pokud konstruktor hodnoty odpovídá vzoru, jsou proměnné vzoru vázány jako u striktního vzoru.
  - Pokud konstruktor hodnoty neodpovídá vzoru, jsou proměnné vzoru vázány na $\bot$.
  - Z hlediska operační sémantiky to znamená, že rozpoznání vzoru je pozdrženo do doby, než je použita nějaká z proměnných.

Nestriktní rozpoznávání vzorů II

Example

$$f :: Bool \to (a, b) \to [a]$$
$$f\ True \sim (a, b) = [\,]$$
$$f\ False\ \ (a, b) = [a]$$

pak

$$f\ True\ \bot = [\,]$$

kdežto kdyby byla f definována klasicky (striktním rozpoznáváním), bylo by

$$f\ True\ \bot = \bot$$

Osnova

Osnova

Cyklické struktury z rekurzivních datových typů

AKA *Tying the knot*

Líné vyhodnocování umožňuje použít strukturu uvnitř její definice.

Example

$$cyclic :: [Int]$$
$$cyclic = 1 : 2 : 3 : cyclic$$

Poznámka: Tato funkce je *korekurzivní*, viz. str. **??**.

Příklad – dvojitý spojový seznam I

```
data DList a = DList (DList a) a (DList a)

singleton :: a → DList a
singleton x = l
  where l = DList l x l

fromList :: [a] → DList a
fromList xs = start
  where
    (start, end)    = f end xs
    f :: DList a → [a] → (DList a, DList a)
    f prev []       = (start, prev)
    f prev (x : xs) = let (next, last) = f this xs
                          this         = DList prev x next
                      in (this, last)
```

Příklad – dvojitý spojový seznam II

```
toRList :: DList a → [a]
toRList (DList _ v n) = v : toRList n

toLList :: DList a → [a]
toLList (DList p v _) = v : toLList p
```

Osnova

Literatura I

[1] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/papers.html.