KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 7: Akumulace

Učební materiál k přednášce 16. listopadu 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2006

Lekce 7: Akumulace

Obsah lekce: V této lekci se budeme zabývat akumulací, což je speciální postupná aplikace procedur. Pomocí akumulace ukážeme efektivnější řešení vybraných úkolů, které jsme řešili v předchozích lekcích, například mapování, filtrace a nahrazování prvků seznamu. Dále ukážeme, jak lze pomocí akumulace rozšířit některé procedury dvou argumentů tak, aby pracovaly s libovolným počtem argumentů.

Klíčová slova: akumulace, filtrace, procedura `foldl`, procedura `foldr`.

7.1 Procedura FOLDR

V této sekci se budeme zabývat *akumulací prvků seznamu*. Pod pojmem *akumulace* máme obvykle na mysli vytvoření jedné hodnoty pomocí více hodnot obsažených v seznamu (sečtení čísel v seznamu, nalezení maxima, vytvoření seznamu pouze z některých hodnot a podobně). Akumulace má blízko k explicitní aplikaci prováděné pomocí primitivní procedury `apply`, o které jsme se bavili v předchozí lekci. Použití `apply` při akumulaci má nevýhodu v tom, že při neznámé délce seznamu můžeme aplikovat pouze procedury s libovolnými argumenty. Jedině tak je totiž při použití `apply` zaručeno, že nedojde k chybě vlivem předání špatného počtu argumentů. Nyní budeme k problému akumulace přistupovat jinak. Seznam libovolné délky budeme akumulovat pomocí *procedur dvou argumentů*, které budou *aplikovány postupně* pro jednotlivé prvky seznamu a to buď zleva nebo zprava.

Nejprve si problém demonstrujeme na příkladu. Uvažujme seznam čísel

`(1 2 3 4)`.

Z předchozích lekcí víme, že tento seznam lze zkonstruovat pomocí konstruktoru párů `cons` a pomocí prázdného seznamu vyhodnocením následujícího výrazu:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))  ⟼  (1 2 3 4)
```

Ve výrazu si můžeme všimnout toho, že se na sebe postupně „nabalují“ výsledky aplikace procedury `cons`. Hodnota vzniklá vyhodnocením `(cons 4 '())`, což je jednoprvkový seznam obsahující čtyřku je dále použita jako druhý argument při další aplikaci `cons` spolu s argumentem `3`, výsledek této aplikace je opět použit při další aplikci `cons` jako druhý argument, a tak dále. Analogické postupné „nabalování“ výsledků aplikací bychom použili, kdybychom chtěli sečíst prvky seznamu za předpokladu, že procedura `+` by akceptovala pouze dva argumenty:

```
(+ 1 (+ 2 (+ 3 4)))
```

Aby podobnost obou výrazů více vynikla, přepíšeme předchozí s využitím součtu s nulou:

```
(+ 1 (+ 2 (+ 3 (+ 4 0))))
```

V podobném stylu bychom mohli vyjádřit součin prvků seznamu:

```
(* 1 (* 2 (* 3 (* 4 1))))
```

Nyní uvedeme ještě jeden příklad, ve kterém pro změnu nebudeme používat při postupném „nabalování“ výsledných hodnot primitivní procedury jako dosud (procedury `cons`, `+` a `*`), ale následující uživatelsky definovanou proceduru:

```
(define y+1
  (lambda (x y)
    (+ y 1)))
```

Všimněte si, že předchozí procedura zcela ignoruje svůj první argument a vrací hodnotu druhého argumentu zvětšenou o jedna (druhý argument tedy musí být číslo). V následující ukázce jsou zachyceny výsledky aplikací této procedury:

```
(y+1 'a 0)                          ⟼  1
(y+1 'a (y+1 'b 0))                 ⟼  2
(y+1 'a (y+1 'b (y+1 'c 0)))        ⟼  3
(y+1 'a (y+1 'b (y+1 'c (y+1 'd 0)))) ⟼  4
```

Jak je asi z ukázky a z definice procedury `y+1` patrné, proceduru můžeme tímto způsobem použít k počítání počtu prvků seznamu. Pro náš výchozí seznam bychom tedy měli:

```
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))  ⟼  4
```

Výraz je opět ve tvaru postupného „nabalování“ aplikací. V čem se lišily všechny předchozí ukázky postupně vnořených aplikací, které jsme uvedli? Pro zopakování, jednalo se o tyto výrazy:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(+ 1 (+ 2 (+ 3 (+ 4 0))))
(* 1 (* 2 (* 3 (* 4 1))))
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))
```

Z hlediska jejich tvaru, ze lišily jen „nepatrně“, protože je lze všechny chápat jako výrazy tvaru:

(⟨*procedura*⟩ 1 (⟨*procedura*⟩ 2 (⟨*procedura*⟩ 3 (⟨*procedura*⟩ 4 ⟨*terminátor*⟩)))),

kde ⟨*procedura*⟩ je procedura dvou argumentů a ⟨*terminátor*⟩ je element. Vskutku, v prvním případě byla ⟨*procedura*⟩ zastoupena `cons` a terminátor byl `()`. V druhém případě byla ⟨*procedura*⟩ zastoupena `+` a ⟨*terminátor*⟩ byl `0`. V posledním případě byla ⟨*procedura*⟩ zastoupena uživatelsky definovanou procedurou `y+1` a ⟨*terminátor*⟩ byl `0`. Z pohledu tvaru se tedy výrazy příliš neliší. Liší se ale výrazně z pohledu svého významu (výsledných hodnot): vytvoření seznamu, součet hodnot, součin hodnot, výpočet délky (což je v podstatě „počet zanoření“ v daném výrazu).

Doposud jsme všechny úvahy prováděli nad seznamem pevné délky. Úvahy bychom ale mohli rozšířit na seznamy libovolných délek. Nabízí se mít k dispozici obecnou proceduru, která pro danou *proceduru dvou argumentů*, *terminátor* a *seznam* provede postupnou aplikaci dané procedury dvou argumentů přes všechny prvky seznamu tak, jak jsme nyní v několika případech ukázali. V jazyku Scheme budeme uvažovat proceduru `foldr` (z anglického *fold right*, neboli „zabal směrem doprava“), která je ve své základní podobě aplikována ve tvaru:

(`foldr` ⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*seznam*⟩).

Při své aplikaci procedura `foldr` provede postupnou aplikaci

(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ ⋯ (⟨*procedura*⟩ ⟨$prvek_n$⟩ ⟨*terminátor*⟩) ⋯))),

pro ⟨*seznam*⟩ ve tvaru (⟨$prvek_1$⟩ ⟨$prvek_2$⟩ ⋯ ⟨$prvek_n$⟩). To jest, použitím `foldr` na prázdný seznam je vrácen element označený jako ⟨*terminátor*⟩. Použitím `foldr` na jednoprvkový, dvouprvkový, tříprvkový, čtyřprvkový (a tak dále) seznam je vrácena hodnota aplikace:

(⟨*procedura*⟩ ⟨$prvek_1$⟩ ⟨*terminátor*⟩)
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ ⟨*terminátor*⟩))
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ ⟨$prvek_3$⟩ ⟨*terminátor*⟩)))
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ ⟨$prvek_3$⟩ (⟨*procedura*⟩ ⟨$prvek_4$⟩ ⟨*terminátor*⟩))))
⋮

Předchozí příklady bychom tedy pomocí `foldr` mohli vyřešit takto:

```
(define s '(1 2 3 4))
(foldr cons '() s)      ⟼  (1 2 3 4)
(foldr + 0 s)           ⟼  10
(foldr * 1 s)           ⟼  24
(foldr y+1 0 s)         ⟼  4
```

První aplikací `foldr` jsme vytvořili duplikát výchozího seznamu, následuje součet a součin prvků seznamu a poslední příklad bychom mohli v souladu s našimi předchozími pozorováními charakterizovat jako vypočtení délky seznamu. Použití `foldr` má zjevnou výhodu v tom, že funguje pro libovolně dlouhý seznam o jehož transformaci na sérii postupných aplikací se jako programátoři nemusíme starat. O tom se ostatně můžete přesvědčit sami, zkuste několikrát změnit seznam nabázaný na `s` a proveďte výše uvedené aplikace `foldr`.

Poznámka 7.1. Všimněte si, jakou roli mají argumenty procedury, kterou při aplikaci předáváme proceduře `foldr`. První argument této procedury postupně *nabývá hodnot vyskytujících se v seznamu*. Na druhou stranu druhý argument zastupuje element, který *vznikl zabalením hodnot* v seznamu vyskytujících se *za průběžným prvkem*. Například při následující aplikaci `foldr`:

```
(foldr (lambda (x y)
         (list #f x y))
       'base
       '(a b c d))  ⟼  (#f a (#f b (#f c (#f d base))))
```

bude procedura vzniklá vyhodnocením λ-výrazu `(lambda (x y) (list #f x y))` nejprve aplikována s hodnotami `d` (poslední prvek seznamu) a `base` (terminátor). Výsledkem tedy bude seznam `(#f d base)`. V dalším kroku bude procedura aplikována s prvkem `c` (předposlední prvek seznamu) a druhým argumentem bude výsledek zabalení vytvořený v předchozím kroku, tedy seznam `(#f d base)`. Výsledkem aplikace procedury proto bude seznam `(#f c (#f d base))`. V dalším kroku bude procedura aplikována s prvkem `b` a seznamem `(#f c (#f d base))`. Jako výsledek vznikne `(#f b (#f c (#f d base)))`. Konečně v posledním kroku bude procedura aplikována s `a` (první prvek seznamu) a s posledním uvedeným seznamem. Výsledek této poslední aplikace bude vrácen jako výsledek aplikace `foldr`.

Následující příklady ukazují roli *terminátoru*.

```
(foldr cons 'base s)     ⟼  (1 2 3 4 . base)
(foldr cons '() s)       ⟼  (1 2 3 4)
(foldr list 'base s)     ⟼  (1 (2 (3 (4 base))))
(foldr list '() s)       ⟼  (1 (2 (3 (4 ()))))
(foldr list '() '())     ⟼  ()
(foldr list 'base '())   ⟼  base
(foldr list 666 '())     ⟼  666
```

Například na prvních dvou použitích `foldr` je vidět, že v prvním případě je aplikována procedura `cons` s argumenty `4` (poslední prvek seznamu) a `base`, což vede k vytvoření tečkového páru `(4 . base)`, který se ve výsledku vyskytuje na konci vytvořené hierarchické struktury. V druhém případě je aplikace `cons` provedena s hodnotami `4` a `()`, což vede na vytvoření jednoprvkového seznamu `(4)` – výsledná struktura vytvořená aplikací `foldr` je v tomto případě seznam. V posledních třech příkladech vidíme mezní případ: při pokusu o zabalení prázdného seznamu je vrácen terminátor.

Pomocí procedury `foldr` lze efektivně implementovat řadu operací nad seznamy. Klíčem ke správnému použití `foldr` je pochopit roli procedury dvou argumentů, která je při aplikaci předávána jako první argument, a pomocí niž je provedeno samotné „zabalení hodnot“. Musíme si uvědomit, že tato procedura je postupně aplikována tak, že její první argument je *průběžný prvek seznamu* a druhý argument je *výsledek zabalení prvků nacházejících se v seznamu za průběžným prvkem*.

Proceduru `foldr` lze použít v obecnějším tvaru. Podobně jako tomu bylo u procedury `map`, procedura `foldr` připouští při aplikaci i více seznamů než jen jeden. Proceduru `foldr` lze tedy aplikovat ve tvaru

`(foldr` ⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*seznam*$_1$⟩ ⟨*seznam*$_2$⟩ $\cdots$ ⟨*seznam*$_n$⟩`)`,

kde ⟨*seznam*$_1$⟩, …, ⟨*seznam*$_n$⟩ jsou seznamy ($n \geq 1$), a ⟨*procedura*⟩ je procedura $n+1$ argumentů. Při aplikaci `foldr` je provedeno analogické zabalení jako ve verzi s jedním seznamem, rozdíl je pouze v tom, že ⟨*procedura*⟩ je aplikována s $n+1$ argumenty, jimiž je n průběžných prvků z předaných seznamů a posledním argumentem je výsledek zabalení prvků následujících za průběžnými prvky.

Nejlépe činnost `foldr` pro víc argumentů vysvětlí následující příklady:

```
(foldr list 'base
       '(1 2 3) '(a b c) '(i j k))   ⟼  (1 a i (2 b j (3 c k base)))
```

```
(foldr (lambda (x y z result)
         (cons (list x y z) result))
       '()
       '(1 2 3) '(a b c) '(i j k))    ⟹  ((1 a i) (2 b j) (3 c k))
```

V následující sekci uvedeme praktické příklady použití `foldr`.

7.2 Použití FOLDR pro definici efektivních procedur

Nyní ukážeme implementaci několika procedur pro práci se seznamy, které už jsme představili v předchozích lekcích. Uvidíme, že vytvoření těchto procedur pomocí `foldr` bude nejen kratší z hlediska jejich zápisu, ale procedury budou mnohem *efektivnější.* I když není výpočtová efektivita hlavním předmětem, na který se v tomto kurzu soustředíme, u každé vytvářené procedury je vždy vhodné zamýšlet se nad její efektivitou. Pro programování ve funkcionálních jazycích je typické vyvíjet program v několika etapách. V první fázi vývoj jde vlastně o obohacení jazyka o nové procedury, pomocí nichž budeme schopni vyřešit daný problém. Po vyřešení problému se můžeme opět vrátit k naprogramovaným procedurám a pokoušet se zvýšit jejich efektivitu tím, že je implementujeme znovu, samozřejmě při zachování dosavadní funkčnosti.

Tento pohled uplatníme v malém měřítku i nyní. Ukážeme si, jak lze efektivně vytvořit procedury, které jsme již měli (méně efektivně) vytvořené. První z nich bude procedura `length` vracející délku seznamu. Původní kód, který jsme vytvořili pomocí `map` a `apply` je uveden v programu 6.1 na straně 144. Před uvedením nové verze `length` se nejdřív zamysleme nad efektivitou původní verze z programu 6.1. Předně, jak můžeme snadno kvantitativně vyjádřit efektivitu procedury pracující se seznamy? Seznamy jsou sekvenční lineární dynamické datové struktury, k jejichž (dalším) prvkům přistupujeme pomocí `cdr`. Tato operace má vzhledem k dalším používaným operacím největší vliv na rychlost zpracování seznamu. Proto budeme vyjadřovat efektivitu *amortizovaně vzhledem k počtu operací* `cdr` provedených nad vstupními daty. Budeme přitom provádět běžná zjednodušení, která jsou studentům známá z kurzu *algoritmické matematiky,* nebudeme se zabývat samotnou strukturou seznamů, ale efektivitu (časovou složitost procedur) budeme vyjadřovat *vzhledem k délce vstupních seznamů.*

Procedura `length` v programu 6.1 počítá výslednou hodnotu tak, že nejprve provede mapování přes celý vstupní seznam. Pokud délku vstupního seznamu označíme n, pak tato fáze zabere n kroků (je potřeba n aplikací `cdr` na průchod seznamem[10]). V další fázi je na vzniklý seznam aplikována operace sčítání ta potřebuje opět n kroků k tomu, aby prošla prvky seznamu (jedničky) a sečetla je. Dohromady tedy procedura pro seznam délky n provede $2n$ kroků. Časovou složitost budeme dále zapisovat v běžné O-notaci, v případě naší procedury tedy $O(2n)$. Intuitivně bychom očekávali, že délku n prvkového seznamu bychom měli stanovit právě v n krocích, procedura `length` z programu 6.1 tedy není příliš efektivní. Na druhou strany není těžké nahlédnout, že délku seznamu se nám nepodaří stanovit v „sublineárním čase“, protože pro výpočet délky seznamu musíme skutečně každý prvek seznamu navštívit aspoň jednou.

Podívejme se nyní na proceduru `length` z programu 7.1. Jde v podstatě jen o formalizaci myšlenky z před-

Program 7.1. Výpočet délky seznamu pomocí `foldr`.

```
(define length
  (lambda (l)
    (foldr (lambda (x y)
             (+ 1 y))
           0 l)))
```

[10] Někdo by v tuto chvíli mohl namítnout, že kroků je potřeba pouze $n - 1$, protože u jednoprvkového seznamu už žádný další prvek nehledáme. Tato úvaha ale není správná, abychom zjistili, že se skutečně jedná o jednoprvkový seznam, musíme otestovat, jaký element se nachází na druhé pozici páru, tím pádem `cdr` musíme skutečně aplikovat n-krát. Uvědomte si, že interpret nevidí seznam „z vnějšku“ tak, jako jej vidíme my.

chozí sekce. Jelikož je v těle procedury aplikována procedura `foldr` se vstupním seznamem, její činnost zabere právě n kroků, během kterých jsou navštíveny všechny prvky seznamu. Hodnota terminátoru `0` znamená, že prázdný seznam má délku nula. Během akumulace je sečteno právě tolik jedniček, kolik je navštíveno prvků, výsledná hodnota je tedy číslo – délka seznamu. Celková časová složitost našeho řešení je tedy $O(n)$.

Z kurzu algoritmické matematiky možná víte, že složitost se stanovuje řádově. Z tohoto pohledu jsou složitosti $O(2n)$ a $O(n)$ obě stejného řádu (lineární složitost), protože multiplikativní konstanta je z hlediska řádové složitosti zanedbatelná[11]. Zde upozorněme na to, že orientovat se podle řádové složitosti, používané třeba ke klasifikaci řešitelných problémů podle jejich časové nebo prostorové složitosti, by bylo z našeho pohledu dost ošidné. Multiplikativní konstanta 2 je z pohledu procedury jako je `length` (která bude v programu zřejmě často používána), dost kritická a naše nová implementace mající složitost $O(n)$ je výrazně lepší než původní se složitostí $O(2n)$[12]. Další nově naprogramovanou procedurou bude spojení dvou seznamů. Původní kód procedury `append2` je k dispozici v programu 5.2 na straně 123. Tato implementace využívající `build-list` je extrémně neefektivní. Označíme-li délku prvního seznamu n a druhého m, pak je nejprve spotřebováno $n+m$ kroků na stanovení délek obou seznamů. Potom je konstruován nový seznam délky $n+m$. Proto, abychom zjistili složitost konstrukce, musíme rozebrat tělo procedury volané procedurou `build-list`. Při pohledu na tělo je jasné, že jsou postupně vraceny prvky z obou seznamů pomocí `list-ref`. Samotná procedura `list-ref` vrátí prvek na k-té pozici (nejdřív) během k kroků. Pro vrácení všech prvků z prvního seznamu tedy potřebujeme $1+2+\cdots+n$ kroků, což je (sečtením prvků aritmetické posloupnosti) dohromady $\frac{n(1+n)}{2}$ kroků. Pro druhý seznam potřebujeme analogicky $\frac{m(1+m)}{2}$ kroků. Celkovou složitost stanovíme součtem všech tří částí (výpočet délek a sekvenční přístup ke všem prvkům obou seznamů), to jest

$$O\big(n+m+\tfrac{n(1+n)}{2}+\tfrac{m(1+m)}{2}\big),$$

což je ekvivalentní

$$O\big(\tfrac{n(n+3)+m(m+3)}{2}\big).$$

Řádově je tedy časová složitost procedury z programu 5.2 dokonce *kvadratická*. To je přímo tristní, protože pro spojení seznamu délky m a seznamu délky n bychom intuitivně očekávali složitost $O(m+n)$. Program 7.2 dokonce ukazuje, že na tom můžeme být ještě o něco lépe. Nejdříve objasněme tělo nové

Program 7.2. Spojení dvou seznamů pomocí `foldr`.

```
(define append2
  (lambda (l1 l2)
    (foldr cons l2 l1)))
```

implementace procedury `append2`. Jedná se o akumulaci prvků prvního seznamu pomocí `cons`, která je terminována druhým předaným seznamem. Všechny prvky prvního seznamu jsou při této akumulaci postupně navštíveny (jeden po druhém), což zabere n kroků. Výsledná složitost je tedy $O(n)$. Pro někoho poněkud překvapivě, protože se do složitosti vůbec nepromítla délka druhého seznamu. Vskutku, druhý seznam je použit jako terminující element a není tedy vůbec procházen. Činnost `append2` napsané pomocí `foldr` si možná lépe uvědomíme, když si představíme, že `foldr` provede sérii aplikací typu:

`(cons` ⟨*prvek*$_1$⟩ `(cons` ⟨*prvek*$_2$⟩ `(cons` $\cdots$ `(cons` ⟨*prvek*$_n$⟩ ⟨*seznam*⟩`)` $\cdots$`)))`,

která skutečně vede na spojení dvou seznamů: `(`⟨*prvek*$_1$⟩ $\cdots$ ⟨*prvek*$_n$⟩`)` a ⟨*seznam*⟩.

Pomocí `foldr` a `append2` můžeme efektivně naprogramovat spojení libovolného množství seznamů tak, jak to ukazuje procedura `append` v programu 7.3. Proceduru jsme pochopitelně museli definovat jako

[11] Z prektického pohledu bychom se měli zajímat i o multiplikativní konstanty zvlášť v případě, pokud jsou velké.

[12] Zde opět připomeňme, že řádově jsou obě složitostní třídy stejné, to jest $O(n) = O(2n)$. Z praktického hlediska je však v tomto případě konstanta 2 výraznou přítěží. Pokud budeme chtít multiplikativní konstanty zdůrazňovat, budeme je v O-notacích uvádět, i když to není běžné.

Program 7.3. Spojení libovolného počtu seznamů pomocí `foldr`.

```
(define append
  (lambda lists
    (foldr append2 '() lists)))
```

proceduru s libovolnými argumenty, ty budou při její aplikaci navázané na symbol `lists`. V těle procedury je použito jedno volání `foldr` pomocí nějž provádíme akumulaci procedury `append2`, kterou jsme vytvořili v předchozím kroku. Tato procedura bude akumulovat prvky ze seznamu `lists`, což jsou seznamy předané proceduře `append` při její aplikaci. Terminátorem je prázdný seznam, protože spojením „žádných seznamů“ vzniká prázdný seznam. Složitost této procedury je rovna $O(n)$, kde n je součet délek všech vstupních seznamů. Pokud použijeme `append` na spojení pouze dvou seznamů, bude mít složitost $O(n+m)$, což je zhoršení oproti `append2`. Důvodem je fakt, že `append` terminuje spojení prázdným seznamem a prochází prvky *všech* předaných seznamů (tedy i toho posledního, v našem případě druhého). To je jakási daň, kterou jsme zaplatili za obecnost řešení.

V programu 5.3 na straně 125 jsme ukázali implementaci procedury `map1`, což je varianta `map` pracující pouze s jedním seznamem. Tato implementace byla opět velmi neefektivní a používala `build-list` a sekvenční vyhledávání prvků pomocí opakované aplikaci `list-ref`. Časová složitost této procedury byla $O\left(\frac{n(n+3)}{2}\right)$, protože n kroků bylo spotřebováno vypočtením délky seznamu a $1+2+\cdots+n$ kroků bylo potřeba na procházení jeho prvků. Složitost takto napsané `map1` byla opět kvadratická. Proceduru lze ale vytvořit se složitostí $O(n)$ tak, jak je to ukázáno v programu 7.4. V nové implementaci `map1` jsme prováděli akumulaci

Program 7.4. Mapovací procedura pracující s jedním seznamem pomocí `foldr`.

```
(define map1
  (lambda (f l)
    (foldr (lambda (x y)
             (cons (f x) y))
           '()
           l)))
```

hodnot pomocí uživatelsky definované procedury, která místo prostého použití `cons` tak, jak jsme jej použili v případě `append2`, provede napojení *modifikace prvního prvku* s již zpracovanou částí. Akumulace je terminována prázdným seznamem. Pomocí `foldr` můžeme nyní naprogramovat i obecný `map` pracující s libovolným (ale nenulovým) počtem seznamů. Obecná verze `map` se nachází v programu 7.5. Program 7.5 obsahuje pomocnou proceduru `separate-last-argument`, která má za účel pro daný neprázdný seznam vrátit tečkový pár, jehož prvním prvkem bude seznam prvků z původního seznamu kromě posledního a druhým prvkem tečkového páru bude poslední prvek seznamu. Viz následující příklady použití:

```
(separate-last-argument '())          ⟹  #f
(separate-last-argument '(a))         ⟹  (() . a)
(separate-last-argument '(a b))       ⟹  ((a) . b)
(separate-last-argument '(a b c))     ⟹  ((a b) . c)
(separate-last-argument '(a b c d))   ⟹  ((a b c) . d)
```

Procedura `separate-last-argument` nám tedy umožňuje přistoupit k prvkům seznamu (vyjma posledního) a k poslednímu prvku. Jedná se tedy o jakési „`car` a `cdr` naruby“. Tuto pomocnou proceduru jsme v programu 7.5 dále použili na implementaci `map`. Samotný `map` jsme realizovali aplikací `foldr`. Jelikož však dopředu nevíme, kolik seznamů bude proceduře `map` předáno, museli jsme proceduru předanou `foldr` vytvořit jako proceduru s libovolným počtem argumentů. Pro n seznamů bude argumentů $n+1$, prvních n argumentů bude reprezentovat průběžné prvky seznamů a poslední argument bude zastupovat

Program 7.5. Obecná mapovací procedura pomocí `foldr`.

```
(define separate-last-argument
  (lambda (l)
    (foldr (lambda (x y)
             (if (not y)
                 (cons '() x)
                 (cons (cons x (car y)) (cdr y))))
           #f
           l)))

(define map
  (lambda (f . lists)
    (apply foldr
           (lambda args
             (let ((separation (separate-last-argument args)))
               (cons (apply f (car separation))
                     (cdr separation))))
           '()
           lists)))
```

akumulovanou hodnotu. Jelikož chceme k akumulované hodnotě přidat hodnotu vzniklou aplikací prvních n argumentů, potřebujeme od sebe nutně oddělit prvních n argumentů a poslední argument. K tomu jsme použili právě pomocnou proceduru `separate-last-argument`, která byla rovněž vytvořená pomocí `foldr`.

Všimněte si, že v proceduře `separate-last-argument` je `foldr` terminován elementem `#f`. V těle procedury předané `foldr` je vidět, že hned při první aplikaci, kdy je na `x` navázaný poslední prvek seznamu, je vytvořen pár ve tvaru `(() . ⟨poslední⟩)`. V každém dalším kroku již se druhý prvek tohoto páru nemění, a do prvního prvku se přidávají postupně procházené prvky. Tím vytvoříme požadovaný výstup, viz výše uvedené příklady.

Složitost obecného `map` můžeme stanovit zhruba takto. Procházíme m seznamů délky n postupně prvek po prvku, to zabere celkem mn kroků. K tomu musíme připočíst režii spojenou s násobnou aplikací pomocné procedury `separate-last-argument`. Tato procedura je aplikována právě tolikrát, jaká je délka seznamů, tedy n-krát. Při každé aplikaci potřebuje $m+1$ kroků na separaci posledního argumentu. Celková složitost je tedy $O(mn + (m+1) \cdot n)$, což je ekvivalentní $O((2m+1) \cdot n)$.

Program 7.6 obsahuje efektivní implementaci filtrační procedury `filter`, kterou jsme představili v programu 6.2 na straně 146. Původní filtrační procedura měla časovou složitost $O(2n)$, zdůvodnění je analogické tomu, jaké jsme provedli u původní procedury `length`. Efektivní implementace z programu 7.6 ukazuje další použití `foldr`. V tomto případě je terminátorem opět prázdný seznam a uživatelsky definovaná procedura předaná `foldr` nejprve otestuje, zda-li průběžný prvek splňuje vlastnost danou procedurou navázanou na symbol `f`. Pokud ano, je prvek přidán k seznamu v němž se akumulují prvky splňující tyto vlastnost. V opačném případě není seznam akumulovaných prvků změněn. Časová složitost nového provedení `filter` je $O(n)$.

Další ukázkou je efektivní implementace predikátu `member?`. Tento predikát jsme představili v programu 6.3 na straně 146. Složitost původní implementace byla $O(2n)$, protože byla založena na původní implementaci `filter`. Kdybychom nyní uvažovali, že ponecháme původní kód `member?`, ale budeme v něm používat novou implementaci `filter` z programu 7.6, pak bude mít `member?` časovou složitost $O(n)$. Můžeme ale provést úplně novou implementaci `member?` přímo použitím `foldr` bez vazby na `filter`. Viz program 7.7.

Poslední procedurou, kterou v této sekci ukážeme je `replace`, která při své aplikaci vyžaduje tři argu-

Program 7.6. Filtrace prvků seznamu splňujících danou vlastnost pomocí `foldr`.

```
(define filter
  (lambda (f l)
    (foldr (lambda (x y)
             (if (f x)
                 (cons x y)
                 y))
           '()
           l)))
```

Program 7.7. Test přítomnosti prvku v seznamu pomocí `foldr`.

```
(define member?
  (lambda (elem l)
    (foldr (lambda (x y)
             (if (equal? x elem) #t y))
           #f
           l)))
```

menty: prvním je predikát jednoho argumentu reprezentující vlastnost prvku seznamu (analogická role jako u `filter`), druhým je procedura jednoho argumentu sloužící k modifikaci prvků seznamu (analogická role jako u `map`) a třetím argumentem je seznam. Výsledkem aplikace procedury `replace` je seznam elementů vzniklý ze vstupního seznamu tak, že každý prvek seznamu splňující vlastnost danou prvním argumentem je modifikován pomocí procedury dané druhým argumentem. Viz program 7.8.

Program 7.8. Nahrazení prvku dané vlastnosti modifikací prvku pomocí `foldr`.

```
(define replace
  (lambda (prop? modifier l)
    (foldr (lambda (x y)
             (if (prop? x)
                 (cons (modifier x) y)
                 (cons x y)))
           '()
           l)))
```

Následující příklady ukazují použití `replace`:

```
(define s '(1 2 3 4 5))
(replace even? (lambda (x) (- x)) s)          ⟼  (1 -2 3 -4 5)
(replace (lambda (x) #t) (lambda (x) 1) s)    ⟼  (1 1 1 1 1)
(replace (lambda (x) (<= x 3)) list s)        ⟼  ((1) (2) (3) 4 5)
(replace (lambda (x) (= 1 (modulo x 3)))
         (lambda (x) (+ x 10))
         s)                                   ⟼  (11 2 3 14 5)
```

7.3 Další příklady akumulace

V této sekci si ukážeme další příklady akumulace pomocí `foldr`. Nejprve se budeme zabývat problematikou rozšíření operace (procedury) dvou argumentů na proceduru *libovolných argumentů*. Konkrétně se budeme zabývat touto problematikou u monoidálních operací, viz sekci 2.5. Pokud je totiž operace $\circledcirc$ na dané množině asociativní a má neutrální prvek, pak můžeme bez újmy psát

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n,$$

protože díky asociativitě nezáleží na uzávorkování předchozího výrazu. Díky neutralitě navíc platí, že

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n \circledcirc e,$$

kde e je neutrální prvek vzhledem k operaci $\circledcirc$. Z hlediska akumulace pomocí `foldr` je pro nás zajímavé

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = (a_1 \circledcirc (a_2 \circledcirc \cdots (a_n \circledcirc e) \cdots)).$$

Pravá strana předchozí rovnosti je ve tvaru vhodném pro akumulaci pomocí `foldr`, protože monoidální operace $\circledcirc$ zde hraje analogickou roli jako procedura předávaná `foldr` a terminátor je neutrální prvek e. Kdybychom tedy ve Scheme neměli k dispozici `+`, `*` a podobné operace jako procedury libovolných argumentů, ale pouze dvou, pak bychom je pomocí `foldr` mohli snadno rozšířit na procedury libovolných argumentů.

V následujícím příkladu máme definovány procedury, které provádějí součet a součin dvou prvků:

```
(define add2 (lambda (x y) (+ x y)))
(define mul2 (lambda (x y) (* x y)))
```

Pomocí `foldr` je můžeme zobecnit na operace pro libovolný počet argumentů:

```
(define ++
  (lambda args
    (foldr add2 0 args)))

(define **
  (lambda args
    (foldr mul2 1 args)))
```

Samozřejmě, že předchozí příklad byl pouze „školský“, protože v interpretu máme k dispozici `+` a `*` pracující s libovolnými argumenty, takže není potřeba je „redukovat na dva argumenty“ pak „opět vyrábět“. Příklad měl sloužit pro demonstraci této obecné techniky. Analogicky jako v předchozím případě bychom mohli na libovolný počet argumentů zobecnit proceduru pro sčítání vektorů pevné délky:

```
(define vec+ (lambda (v1 v2) (map + v1 v2)))
```

Zde je ale malý problém s neutrálním prvkem. Neutrální prvek pro sčítání vektorů je pochopitelně nulový vektor. Pokud jsou vektory reprezentovány seznamem hodnot, pak by to měl být seznam skládající se ze samých nul. Potíž je ale v tom, že předchozí procedura byla schopná sčítat vektory *libovolné délky*, pro každou z délek máme jeden neutrální prvek. Situaci bychom mohli vyřešit tak, že bychom vytvořili proceduru vyššího řádu `make-vec+`, která by pro danou délku vrátila proceduru pro sčítání libovolně mnoha vektorů:

```
(define make-vec+
  (lambda (n)
    (let ((null-vector (build-list n (lambda (i) 0))))
      (lambda vectors
        (foldr vec+ null-vector vectors)))))
```

Použití procedury by pak bylo následující:

```
((make-vec+ 3))                                  ⟹  (0 0 0)
((make-vec+ 3) '(1 2 3))                         ⟹  (1 2 3)
((make-vec+ 3) '(1 2 3) '(10 20 30))             ⟹  (11 22 33)
((make-vec+ 3) '(1 2 3) '(10 20 30) '(2 4 6))    ⟹  (13 26 39)
```

Proceduru `make-vec+` bychom místo `foldr` a `vec+` mohli implementovat s použitím `map` pro libovolné argumenty. Následující příklad rovněž ukazuje použití `apply` s nepovinnými argumenty.

```
(define make-vec+
  (lambda (n)
    (let ((null-vector (build-list n (lambda (i) 0))))
      (lambda vectors
        (apply map + null-vector vectors)))))
```

Zamysleme se nyní nad možností rozšířit proceduru pro výpočet minima ze dvou prvků na proceduru zpracovávající libovolné argumenty:

```
(define min2
  (lambda (x y)
    (if (<= x y) x y)))
```

Problémem je, že „operace minimum“ se sice chová asociativně, to jest platí

$$\min(a, \min(b, c)) = \min(\min(a, b), c),$$

ale *nemá neutrální prvek.* Nyní máme několik možností, jak postupovat. Jednou z možností je implementovat procedury pro výpočet minima tak, jak je v současném standardu R^{5}RS jazyka Scheme, viz [R5RS]. To jest uvažujeme minimum z jednoho a více čísel:

```
(define min
  (lambda numbers
    (foldr min2 (car numbers) (cdr numbers))))
```

V předchozím kódu jsme jako terminátor zvolili první prvek seznamu čísel, se kterými je procedura `min` aplikována. Samotnou akumulaci pak provádíme přes seznam předaných čísel bez prvního. Zde jsme vlastně tiše využili i komutativitu sčítání čísel, protože pro seznam obsahující hodnoty $a_1, \ldots, a_n$ počítáme výsledek takto:

$$\min(a_2, \min(a_3, \ldots \min(a_n, a_1) \cdots)).$$

Z těla výše uvedené procedury je taky jasné, že aplikace `min` bez argumentu by skončila chybovým hlášením způsobeným použitím `car` a `cdr` na prázdný seznam.

Druhým způsobem řešení problému je neutrální prvek nějak „dodat“. Například bychom mohli uvažovat symbol `+infty`, který by nám zastupoval „plus nekonečno“. Tedy jakési nestandardní „největší číslo“. Tento krok by znamenal upravit predikát `<=` (a v důsledku i další aritmetické procedury) tak, aby pracoval i s touto novou hodnotou. Úprava by mohla vypadat takto, nejprve nadefinujeme `+infty`, který se bude vyhodnocovat na sebe sama:

```
(define +infty '+infty)
```

Dále vytvoříme novou verzi predikátu porovnávání čísel:

```
(define <=
  (let ((<= <=))
    (lambda (x y)
      (or (equal? y +infty)
          (and (not (equal? x +infty))
               (<= x y))))))
```

Nový `<=` se na číselných hodnotách chová stejně jako stará verze, pro `+infty` se chová tak, že `+infty` je „větší než všechno ostatní“. Viz následující příklady použití.

```
(<= 2 3)            ⟼  #t
(<= 3 2)            ⟼  #f
(<= 2 +infty)       ⟼  #t
(<= +infty 3)       ⟼  #f
(<= +infty +infty)  ⟼  #t
```

Nyní můžeme ponechat kód `min2` tak, jak jej máme, a pouze definujeme novou obecnou verzi `min`:

```
(define min
  (lambda numbers
    (foldr min2 +infty numbers)))
```

Takto definovanou proceduru je možné použít běžným způsobem, nyní i bez argumentů:

```
(min)           ⟼  +infty
(min 30)        ⟼  30
(min 30 10)     ⟼  10
(min 30 10 20)  ⟼  10
```

Nyní se vraťme k reprezentaci množin uvedené v sekci 6.5, kde jsme implementovali konstruktor množiny `list->set`. Ten ze seznamu vytvářel množinu tím, že podle tohoto seznamu odstranil duplicitní výskyty prvků. Pomocí procedury `foldr`, můžeme napsat elegantnější řešení:

```
(define list->set
  (lambda (l)
    (foldr (lambda (x y)
             (if (member? x y)
                 y
                 (cons x y)))
           '()
           l)))
```

Takto nadefinovaná procedura `list->set` provádí akumulaci pomocí `foldr` přes zadaný seznam. Jako terminátor je zvolen prázdný seznam. Vstupní procedura procedury `foldr` pak testuje přítomnost průběžného prvku v seznamu, který vznikl v předchozím kroku (používá se zde predikátu `member?`, který jsme napsali v programu 7.7). Pokud zjistí, že průběžný prvek v seznamu ještě není, přidá tento prvek do seznamu. V opačném případě vrací nezměněný seznam. Tak jsou odstraněny duplicitní výskyty, viz příklady použití.

```
(list->set '())               ⟼  ()
(list->set '(1 2 3))          ⟼  (1 2 3)
(list->set '(1 2 2 1 2 3 1))  ⟼  (2 3 1)
```

Teď se budeme zabývat možným zobecněním procedury `foldr`. Jedním z argumentů procedury `foldr` je ⟨*procedura*⟩. Tato procedura ⟨*procedura*⟩ nese vlastně dvě informace. Říká, jakým způsobem se modifikuje *průběžný prvek* a jakým způsobem se tato modifikace „nabalí“ na *zabalení modifikovaných hodnot za průběžným prvkem*. Vzhledem k tomu můžeme tuto proceduru rozdělit na dvě, tak aby každá z nich obsahovala jen jednu z těchto informací. Například:

- V programu 7.4 jsme definovali proceduru mapování přes jeden seznam `map1`. Proceduru `foldr` jsme aplikovali na proceduru, která je výsledkem vyhodnocení λ-výrazu

  `(lambda (x y) (cons (f x) y))`.

  Procedurou modifikující průběžný prvek je v tomto případě vstupní procedura procedury `map1` navázaná na symbol `f`. Výsledek aplikace této procedury na průběžný prvek pak kombinujeme se zbytkem pomocí konstruktoru `cons`.

- V programu 7.1, ve kterém jsme definovali proceduru `lenght`, předáváme proceduře `foldr` proceduru, která vznikne vyhodnocením výrazu `(lambda (x y) (+ 1 y))`. Argument `x` je v ní ignorován a je místo něj uvažováno číslo `1`. A toto číslo je přičítáno k zabalení zbytku. Rozdělit bychom ji mohli na konstantní proceduru vracející vždy `1` a na primitivní proceduru sčítání navázanou na symbol `+`.

Toto zobecnění napíšeme s pomocí procedury `foldr`. Procedura bude brát čtyři argumenty. Prvním z nich bude procedura `combinator` o dvou argumentech, která bude určovat způsob nabalování. Smysl těchto argumentů je v podstatě stejný jako u procedury, která je argumentem procedury `foldr`. Rozdíl je jen v tom, že jí jako první argument není předáván přímo průběžný prvek, ale jeho modifikace. Modifikací myslíme

výsledek aplikace procedury `modifier`, která je druhým argumentem procedury `accum`, na průběžný prvek.

```
(define accum
  (lambda (combinator modifier nil l)
    (foldr (lambda (x y)
             (combinator (modifier x) y))
           nil l)))
```

Uvádíme několik příkladů volání této akumulační procedury:

```
(accum + (lambda (x) x) 0 '(1 2 3 4))          ⟹  10
(accum + (lambda (x) (* x x)) 0 '(1 2 3 4))    ⟹  30
(accum + (lambda (x) 1) 0 '(1 2 3 4))          ⟹  4
```

Už jsme naznačili, jak by se pomocí této obecné akumulační procedury daly napsat procedury `map1`a `lenght`. Na závěr ještě ukážeme, jak bychom ji mohli použít k filtrování seznamu. Kombinační procedurou bude spojování seznamu `append`, procedurou modifikující průběžné prvky bude procedura, která v závislosti na platnosti vstupního predikátu vrací buďto prázdný seznam `()`, nebo jednoprvkový seznam obsahující tento průběžný prvek. Jako terminátor použijeme prázdný seznam. Následuje celý kód:

```
(define filter
  (lambda (f l)
    (accum append
           (lambda (x)
             (if (f x)
                 (list x)
                 '()))
           '()
           l)))
```

7.4 Procedura FOLDL

V této sekci se zaměříme na variantu akumulační procedury `foldr`. Jak jsme si již mohli všimnout, `foldr` pracoval tím způsobem, že provedl sérii aplikací procedury dvou argumentů, čímž nám umožnil postupně na sebe „nabalovat“ výsledky aplikací. Toto „nabalování“ přitom postupovalo směrem doprava:

(⟨*procedura*⟩ ⟨*prvek*$_1$⟩ (⟨*procedura*⟩ ⟨*prvek*$_2$⟩ (⟨*procedura*⟩ $\cdots$ (⟨*procedura*⟩ ⟨*prvek*$_n$⟩ ⟨*terminátor*⟩) $\cdots$))).

První aplikace, která je dokončena, je aplikace provedená s posledním prvkem seznamu. Následuje aplikace provedená nad předposledním prvkem seznamu a tak se postupuje až k prvnímu prvku. Tento proces bychom také mohli obrátit. Mohli bychom uvažovat zabalení v tomto směru:

(⟨*procedura*⟩ (⟨*procedura*⟩ $\cdots$ (⟨*procedura*⟩ (⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*prvek*$_1$⟩) ⟨*prvek*$_2$⟩) $\cdots$) ⟨*prvek*$_n$⟩),

kdy je jako první aplikována procedura na terminátor a první prvek seznamu, výsledek je použit při aplikaci s druhým prvkem seznamu a tak dále. Jako poslední je provedena aplikace s posledním prvkem seznamu. U procedury `foldr` tedy probíhaly aplikace směrem *zprava* (odtud název *fold right*). U nově uvedeného typu „zabalení“ probíhá aplikace směrem *zleva*. Nabízí se tedy uvažovat proceduru vyššího řádku, která by byla duální k `foldr` a prováděla zabalení druhým z uvedených způsobů (zleva). Tuto proceduru nazveme `foldl` (z anglického *fold left*).

Procedura `foldl` bude mít argumenty stejného typu a významu jako měla procedura `foldr`, nebudeme je tedy opakovat. Při své aplikaci provede postupnou sérii aplikací dané procedury na prvky seznamu (směrem zleva), která je ukončena terminátorem. V literatuře [BW88] se lze setkat s různými variantami `foldl`, které se liší tím, jaký význam má první a druhý argument procedury, která je předaná `foldl` jako první argument. Podle [BW88] je výsledkem aplikace

(`foldl` ⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*seznam*⟩)

série aplikací tvaru

$$(\langle procedura\rangle\ (\langle procedura\rangle \cdots (\langle procedura\rangle\ (\langle procedura\rangle\ \langle terminátor\rangle\ \langle prvek_1\rangle)\ \langle prvek_2\rangle) \cdots)\ \langle prvek_n\rangle),$$

to jest přesně tak, jak jsme naznačili v úvodu sekce. Lze také uvažovat sérii aplikací vypadající takto:

$$(\langle procedura\rangle\ \langle prvek_n\rangle\ (\langle procedura\rangle\ \langle prvek_{n-1}\rangle\ (\langle procedura\rangle \cdots (\langle procedura\rangle\ \langle prvek_1\rangle\ \langle terminátor\rangle) \cdots))).$$

Mezi oběma předchozími sériemi aplikací je zcela zřejmě *jediný rozdíl.* V prvním případě je $\langle procedura\rangle$ aplikována tak, že jejím prvním argumentem je výsledek předchozí akumulace a druhým argumentem je průběžný prvek seznamu. V druhém případě je tomu obráceně: prvním argumentem je průběžný prvek a druhým argumentem je výsledek předchozí akumulace. V obou případech je ale akumulace zahájena od prvního prvku seznamu.

Z toho, co jsme teď uvedli, by mělo být zřejmé, že procedury provádějící výše uvedené „zabalení zleva" budeme schopni naprogramovat pomocí `foldr` a to v případě obou typů sérií aplikací. Pro druhý typ je to jednodušší, protože stačí použít `foldr` na převrácený seznam. V případě prvního typu pak už jen stačí obrátit argumenty při aplikaci procedury. Procedury provádějící obě zabalení jsou prezentovány v programu 7.9. Procedura pojmenovaná `genuine-foldl` reprezentuje „zabalení zleva" podle [BW88],

Program 7.9. Procedury `genuine-foldl` a `foldl` vytvořené pomocí `foldr` a reverze seznamu.

```
(define genuine-foldl
  (lambda (f term l)
    (foldr (lambda (x y)
             (f y x))
           term
           (reverse l))))

(define foldl
  (lambda (f term l)
    (foldr f term (reverse l))))
```

tedy první z uvedených typů. Procedura `foldl` reprezentuje druhý z typů. Rozdíly mezi oběma typy zabalení a rozdíl oproti `foldr` si nejlépe uvědomíme na následujícím příkladu. Nejprve nadefinujeme pomocnou proceduru:

```
(define proc
  (lambda (x y)
    (list #f x y))),
```

kterou pak použijeme s týmž seznamem při aplikaci `foldr`, `foldl` a `genuine-foldl`:

```
(define s '(a b c d))
(foldr proc 'base s)          ⟼  (#f a (#f b (#f c (#f d base))))
(foldl proc 'base s)          ⟼  (#f d (#f c (#f b (#f a base))))
(genuine-foldl proc 'base s)  ⟼  (#f (#f (#f (#f base a) b) c) d)
```

Jak vidíme, výsledky aplikaci odpovídají oběma typům, které jsme uvedli v této sekci.

Poznámka 7.2. Jedním ze základních vztahů, který platí mezi `foldr` a `genuine-foldl` je ten, že pokud je při akumulaci použita *monoidální procedura* a jako terminátor je použit její *neutrální prvek*, pak je *výsledek použití* `foldr` *a* `genuine-foldl` *stejný.* Toto pozorování lze jednoduše dokázat.

Jako příklad použití `foldl` si můžeme uvést proceduru `reverse` provádějící otočení seznamu:

```
(define reverse
  (lambda (l)
    (foldl cons '() l)))
```

Tento příklad je poněkud „umělý“, protože v programu 7.9 jsme samotný `foldl` zavedli pomocí `reverse`. Kdybychom to učinili a poté definovali `reverse` předchozím způsobem, při pokusu o jeho aplikaci bychom se dostali do nekončící série aplikací (protože `foldl` aplikuje `reverse` a obráceně). Ukažme tedy o něco přirozenější příklad. V sekci 2.5 jsme představili proceduru vyššího řádu `compose2` vracející, pro dvě vstupní procedury jednoho argumentu, proceduru reprezentující jejich složení, viz příklad 2.6 na straně 60. Nyní bychom mohli pomocí `foldl` naprogramovat složení libovolného množství procedur tak, jak to ukazuje program 7.10. Procedura `compose` je pomocí `foldl` vytvořena přímočaře. Terminátorem je identita,

Program 7.10. Složení libovolného množství procedur pomocí `foldl`.

```
(define compose
  (lambda functions
    (foldl (lambda (f g)
             (lambda (x) (f (g x))))
           (lambda (x) x)
           functions)))
```

což je neutrální prvek vzhledem ke skládání. Procedura dvou argumentů, která je při aplikaci předána `foldl`, provádí složení akumulované hodnoty (procedury vzniklé předchozími složeními) s průběžnou procedurou ze seznamu procedur `functions`. Proč jsme při skládání nepoužili `foldr` jako u všech ostatních procedur v této lekci? Protože jsme chtěli pro seznam procedur reprezentující funkce $f_1, \ldots, f_n$ (v tomto pořadí) vrátit proceduru reprezentující jejich kompozici $f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n$, která je daná

$$(f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n)(x) = (f_n(f_{n-1}(\cdots(f_2(f_1(x)))\cdots))),$$

což vede k použití „zabalení zleva“ – skládání je potřeba aplikovat směrem „zepředu“ seznamu (tedy používáme „zabalení zleva“). Vzhledem k tomu, že skládání funkcí na množině je monoidální operace, použití `genuine-foldl` by nám nepomohlo (vedlo by to na stejný výsledek jako použití `foldr`), protože bychom tím provedli složení v opačném pořadí, což je vzhledem k nekomutativitě skládání funkcí problém.

Následující příklady ukazují použití `compose`. Nejprve použijeme pomocné definice

```
(define s '(0 1 2 3 4))
(define f1 (lambda (x) (* 2 x)))
(define f2 (lambda (x) (* x x)))
(define f3 (lambda (x) (+ x 1))),
```

které dále použijeme při skládání pomocí `compose`:

```
(map (compose) s)           ⟼  (0 1 2 3 4)
(map (compose f1) s)        ⟼  (0 2 4 6 8)
(map (compose f1 f2) s)     ⟼  (0 4 16 36 64)
(map (compose f2 f1) s)     ⟼  (0 2 8 18 32)
(map (compose f1 f2 f3) s)  ⟼  (1 5 17 37 65)
(map (compose f3 f2 f1) s)  ⟼  (2 8 18 32 50)
  ⋮
```

Poznamenejme, že k procedury `genuine-foldl` a `foldl` budeme rovněž chápat jako procedury pracující nad libovolným počtem seznamů (vždy alespoň nad jedním) stejně tak, jako tomu bylo i u procedur `foldr`, `map` a podobně. V programu 7.11 je uvedeno rozšíření těchto procedur tak, aby nepracovaly pouze s jedním seznamem, ale obecně s více seznamy. V obou případech jsme rozšířili seznam argumentů o volitelnou část a v těle jsme provedli explicitní aplikaci `foldr` pomocí `apply`. Následující příklady ukazují činnost obou procedur pro více seznamů:

```
(foldl list 'base '(a b) '(1 2) '(#f #t))           ⟼  (b 2 #t (a 1 #f base))
(genuine-foldl list 'base '(a b) '(1 2) '(#f #t))   ⟼  ((base #f 1 a) #t 2 b)
```

Program 7.11. Procedury `genuine-foldl` a `foldl` pracující s libovolným počtem seznamů.

```
(define genuine-foldl
  (lambda (f term . lists)
    (apply foldr
           (lambda args
             (apply f (reverse args)))
           term
           (map reverse lists))))

(define foldl
  (lambda (f term . lists)
    (apply foldr f term (map reverse lists))))
```

Poznámka 7.3. Z posledního příkladu a z implementace `genuine-list` jsme si mohli všimnout, že proceduru předávanou `genuine-list` jsme aplikovali s převráceným seznamem argumentů. U procedur více než dvou argumentů ale není jasné, zda-li by se toto „převrácení“ mělo týkat všech argumentů nebo jestli bychom pouze neměli dát poslední argument (zastupující akumulovanou hodnotu) na začátek seznamu. Při definici `genuine-foldl` pracující pouze s jedním seznamem jsme tento problém nemuseli vůbec uvažovat. Také si všimněte, že procedury `foldl` se tento problém netýká, protože má argumenty pořád ve stejném pořadí. Z tohoto důvodu budeme dále preferovat používání procedury `foldl` nad procedurou `genuine-foldl` (nemluvě o tom, že ve Scheme má `foldl` praktičtější uplatnění).

Procedury `foldr` a `foldl` nejsou přítomny ve standardu R^{5}RS jazyka Scheme, ačkoliv některé interprety jazyka Scheme jimi disponují. Tyto procedury jsou přítomny v mnoha funkcionálních programovacích jazycích. Ve Scheme si `foldr` i `foldl` můžeme naprogramovat, což ukážeme v dalších lekcích.

7.5 Další příklady na FOLDR a FOLDL

V této sekci uvedeme praktické použití akumulační procedury `foldr`. Jako první se budeme zabývat procedurou, která bere jako argument libovolný seznam a vrací seznam všech jeho suffixů – včetně prázdného. Použijeme proceduru `foldr` tímto způsobem: Procedura, která je jejím prvním argumentem, vybere ze seznamu doposud nalezených suffixů první prvek, přidá do něj průběžný prvek seznamu a výsledný seznam přibalí k seznamu nalezených suffixů. Tento seznam obsahuje z počátku jen prázdný seznam, protože prázdný seznam je suffixem jakéhokoli seznamu. Tím je dán druhý argument procedury `foldr`. Posledním (třetím) argumentem předaným `foldr` je samotný seznam, jehož suffixy hledáme. Implementace by pak vypadala takto:

```
(define suffixes
  (lambda (l)
    (foldr (lambda (x y)
             (cons (cons x (car y)) y))
           '(())
           l)))
```

Aplikací takto nadefinované procedury dostáváme seznam suffixů seznamu seřazené od nejdelšího po nejkratší (to jest po prázdný seznam):

```
(suffixes '())       ⟹  (())
(suffixes '(1))      ⟹  ((1) ())
(suffixes '(1 2))    ⟹  ((1 2) (2) ())
(suffixes '(1 2 3))  ⟹  ((1 2 3) (2 3) (3) ())
```

Pokud bychom chtěli jen neprázdné suffixy, mohli bychom to udělat mnoha způsoby s použitím procedury `suffixes`, kterou jsme právě nadefinovali. Ze seznamu suffixů, který je výsledkem aplikace této procedury, pak můžeme odstranit prázdný seznam vyfiltrováním neprázdných seznamů, odstraněním posledního prvku, a tak dále. Též bychom mohli použít následující elegantní řešení:

```
(define safe-car
  (lambda (x)
    (if (null? x)
        '()
        (car x))))

(define suffixes
  (lambda (l)
    (foldr (lambda (x y)
             (cons (cons x (safe-car y)) y))
           '()
           l)))
```

Uvedený program obsahuje definici bezpečné verze selektoru `car`. Tuto bezpečnější verzi `safe-car` jsme popsal i v sekci 5.4. Jinak se nová procedura `suffixes` liší jen použití procedury `safe-car` namísto `car`, a v použití prázdného seznamu jako terminátoru. Použití `safe-car` je důležité při první aplikaci procedury, která je argumentem `foldr`, kdy je aplikována na prázdný seznam.

Použití této procedury dostáváme podobné výsledky jako dříve. Liší se jen v absenci prázdného seznamu v seznamu nalezených sufixů.

```
(suffixes '())          ⟼  ()
(suffixes '(a))         ⟼  ((a))
(suffixes '(a b c))     ⟼  ((a b c) (b c) (c))
(suffixes '(a b c d))   ⟼  ((a b c d) (b c d) (c d) (d))
```

Kdybychom namísto procedury `foldr` použili proceduru `foldl`, nebyl by výsledkem seznam sufixů, ale naopak seznam prefixů. To je samozřejmě způsobeno změnou směru, kterým je akumulace prováděna.

```
(define prefixes
  (lambda (l)
    (foldl (lambda (x y)
             (cons (append (safe-car y) (list x)) y))
           '()
           l)))
```

Takto nadefinovanou procedurou můžeme hledat všechny neprázdné prefixy zadaného seznamu.

```
(prefixes '())          ⟼  ()
(prefixes '(a))         ⟼  (a)
(prefixes '(a b c))     ⟼  ((a b c) (a b) (a))
(prefixes '(a b c d))   ⟼  ((a b c d) (a b c) (a b) (a))
```

7.6 Výpočet faktoriálu a Fibonacciho čísel

Procedury `foldr` a `foldl` lze použít i pro výpočet hodnot matematických funkcí. V této sekci si ukážeme dvě ukázky použití `foldr` při výpočtu *faktoriálu* a *Fibonacciho čísel*. Hned na počátku však řekněme, že příklady reprezentované v této sekci mají spíš „odstrašující charakter“. Jejich smyslem je poukázat na fakt, že i když se nám podařilo pomocí `foldr` vytvořit řadu užitečných a efektivních procedur (s krátkým a přehledným tělem), ne vždy je použití `foldr` na místě.

Připomeňme, že faktoriál $n!$ nezáporného čísla n je definován jako součin přirozených čísel od 1 do n. Neformálně jej tedy lze chápat jako číslo dané

$$n! = \underbrace{1 \cdot 2 \cdot \cdots \cdot n}_{n \text{ činitelů}}.$$

V další sekci ukážeme zavedení faktoriálu, které je z matematického hlediska přesnější. V tuto chvíli si ale vystačíme s touto poněkud neformální definicí. Pro $n = 0, 1, 2, \ldots$ nabývá faktoriál $n!$ následujících hodnot:

$$1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, \ldots$$

Naším úkolem nyní je naprogramovat proceduru `fac`, která pro daný argument jímž bude nezáporné číslo n, vrátí hodnotu faktoriálu $n!$. Z předchozího je tedy jasné, že naše procedura musí provést součin $1 \cdot 2 \cdot \cdots \cdot n$. Nabízí se tedy pomocí `build-list` nejprve vytvořit seznam činitelů a potom jej vynásobit pomocí `apply` nebo `foldr`. Obě verze jsou uvedeny v programu 7.12. Procedury skutečně počítají to, co

Program 7.12. Výpočet faktoriálu pomocí procedur vyšších řádů.

```
(define fac
  (lambda (n)
    (apply * (build-list n (lambda (x) (+ x 1))))))

(define fac
  (lambda (n)
    (foldr * 1 (build-list n (lambda (x) (+ x 1))))))
```

mají, o čemž se můžeme přesvědčit například vyhodnocením následujícího výrazu:

```
(map fac '(0 1 2 3 4 5 6 7 8 9))  ⟹  (1 1 2 6 24 120 720 5040 40320 362880)
```

Mnoha čtenářům by se ale mohlo zdát, že procedura `fac` pracuje až příliš složitě na to, že počítá tak jednoduchou funkci jako je faktoriál. To je pravda. Během výpočtu jsme zkonstruovali pomocný seznam, což trvalo n kroků a jeho prvky jsme posléze vynásobili, to trvalo dalších n kroků. Časová složitost výpočtu tedy byla $O(2n)$, při výpočtu jsme navíc konstruovali seznam, který jsme použili pouze jednorázově. Ani kód procedury nebyl tak čitelný, jak bychom (u jednoduché funkce jakou je faktoriál) očekávali. Lepší variantu procedury `fac` ukážeme v další lekci.

Druhým příkladem v této sekci bude procedura pro výpočet prvků Fibonacciho posloupnosti. Fibonacciho posloupnost je tvořena počátečními dvěma prvky $F_0 = 0$, $F_1 = 1$ a každý další prvek posloupnost vzniká součtem předchozích dvou, posloupnost tedy vypadá následovně:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, \ldots$$

Procedura počítající prvky Fibonacciho posloupnosti je uvedená v programu 7.13. Rozeberme si nyní, jak

Program 7.13. Výpočet prvků Fibinacciho posloupnosti pomocí procedur vyšších řádů.

```
(define fib
  (lambda (n)
    (cadr
     (foldr (lambda (x last)
              (list (apply + last) (car last)))
            '(1 0)
            (build-list n (lambda (x) #f))))))
```

je procedura `fib` naprogramovaná. Procedura ve svém těle provádí akumulaci pomocí `foldr`. Při této akumulaci jsou postupně sčítány dvě předchozí Fibonacciho čísla, čímž se získá další prvek posloupnosti. Ten je dál použit s předchozím prvkem k získání dalšího prvku a tak dále. Akumulace probíhá přes seznam vytvořený aplikací `build-list`. Tento seznam má délku n. Samotné prvky seznamu pro nás nebudou mít žádný význam, seznam je použit pouze jako „čítač kroků“ určující, kolikátý člen Fibonacciho posloupnost

má být nalezen. Při samotné akumulaci je vždy vytvářen dvouprvkový seznam ve tvaru $(F_{i+1}\ \ F_i)$. Tedy druhý prvek seznamu je průběžné Fibonacciho číslo a první prvek seznamu je jeho následník. Pokud máme k dispozici F_i a F_{i+1} můžeme hodnotu F_{i+2} stanovit součtem $F_i + F_{i+1}$, což provádíme při akumulaci, viz explicitní aplikaci procedury sčítání. Výsledkem akumulace v i-tém kroku je tedy vytvoření seznamu tvaru $(F_{i+2}\ \ F_{i+1})$ na základě seznamu $(F_{i+1}\ \ F_i)$, který je navázaný na symbol `last` (všimněte si, že argument `x` je ignorován). Terminátorem akumulace je seznam $(F_1\ \ F_0)$, to jest seznam `(1 0)`. Pro dané n je výsledkem akumulace seznam $(F_{n+1}\ \ F_n)$, stačí tedy vrátit jeho druhý prvek, což je požadovaný výsledek, to je provedeno aplikací `cadr` na výsledek akumulace. Viz příklad použití procedury:

```
(map fib '(0 1 2 3 4 5 6 7 8 9))  ⟹  (0 1 1 2 3 5 8 13 21 34)
```

Při akumulaci se postupně vytváří dvouprvkové seznamy posledních dvou uvažovaných Fibonacciho čísel. Například při výpočtu `(fib 9)` bude argument `last` při aplikaci procedury předané `foldr` postupně nabývat hodnot `(1 0)`, `(1 1)`, `(2 1)`, `(3 2)`, `(5 3)`, `(8 5)`, `(13 8)`, `(21 13)` a konečně `(34 21)`. Výsledkem akumulace bude tím pádem seznam `(55 34)`, jehož druhý prvek je vrácen jako výsledek výchozí aplikace `(fib 9)`. Netřeba asi zdůrazňovat, že tento způsob výpočtu Fibonacciho čísel je opět dost neefektivní. V první řadě, kód procedury `fib` je dost nestravitelný a jeho úplné pochopení již vyžaduje nějakou chvíli. Během výpočtu je dále konstruována celá řada „odpadních seznamů“, například n-prvkový seznam `(#f #f` $\cdots$, přes který se akumuluje, a jehož hodnoty (paradoxně) nemají žádný význam. Dále v každém kroku konstruujeme dvouprvkový seznam udržující informaci o posledních dvou stanovených Fibonacciho číslech, což také výrazně ubírá na efektivitě.

Čistota kódu (jeho jednoduchost a čitelnost) a jeho efektivita (výkon) jdou v některých případech proti sobě. Výše uvedené příklady však nejsou ani čistě provedené ani efektivní. V další sekci se mimo jiné zaměříme na zefektivnění a čistější naprogramování procedur `fac` a `fib`.

Shrnutí

V této lekci jsme ze zabývali akumulací, což je speciální postupná aplikace procedur. V jazyku Scheme jsme uvažovali dodatečné procedury `foldr` a `foldl`, pomocí nichž lze akumulaci provádět. Ukázali jsme efektivní implementace vybraných procedur pracujících se seznamy, například mapování, filtrace a nahrazování prvků seznamu. Provedli jsme diskusi ohledně jejich časové složitosti i ohledně složitosti jejich původních verzí. Dále jsme se zabývali problematikou rozšiřování procedur dvou argumentů tak, aby pracovaly s libovolným počtem argumentů. Lekci jsme zakončili ukázkou metody výpočtu faktoriálu a Fibonacciho čísel pomocí akumulace.

Pojmy k zapamatování

- akumulace, explicitní aplikace, filtrace,
- zabalení směrem doprava, zabalení směrem doleva,
- rozšíření procedur dvou argumentů na libovolné argumenty,
- efektivní implementace procedur,
- výpočet faktoriálu a Fibonacciho čísel

Nově představené prvky jazyka Scheme

- procedury `foldr`, `foldl` a `genuine-foldl`.

Kontrolní otázky

1. *Čím se od sebe liší* `foldr` *a* `foldr`*?*
2. *Jak probíhá aplikace* `foldr`*?*
3. *K čemu slouží terminátory?*
4. *Jak lze využít* `foldr` *k rozšíření procedury dvou argumentů na libovolný počet argumentů?*
5. *Co pro nás hrálo klíčovou roli při stanovování časové náročnosti procedur?*

6. *Jaký je rozdíl mezi* `foldl` *a* `genuine-foldl`*?*

Cvičení

1. V sekci 7.3 jsme rozšířili proceduru `min2` na proceduru libovolného počtu argumentů. Stejným způsobem rozšiřte proceduru na výběr čísla s extrémní absolutní hodnotou `abs-min`.
2. Napište predikát, který pro posloupnost zjistí, zda je neklesající. Posloupnost bude reprezentovaná seznamem, jehož prvky jsou čísla. Viz příklady aplikace:
```
(nondecreasing? '())          ⟹  #t
(nondecreasing? '(1 2 3 4))   ⟹  #t
(nondecreasing? '(1 2 4 3))   ⟹  #f
(nondecreasing? '(1 4 2 3))   ⟹  #f
(nondecreasing? '(4 1 2 3))   ⟹  #f
```
3. Napište procedury `after` a `before`, jejichž argumenty budou element ⟨*elem*⟩ a seznam ⟨*l*⟩. Procedura `after` bude vracet seznam prvků za posledním výskytem prvku ⟨*elem*⟩ (včetně) v seznamu ⟨*l*⟩. Procedura `before` zase seznam prvků před prvním výskytem prvku ⟨*elem*⟩ (včetně) v seznamu ⟨*l*⟩. Viz příklady použití:
```
(after 10 '(1 2 3 4 3 5 6))    ⟹  ()
(after 3 '(1 2 3 4 3 5 6))     ⟹  (3 5 6)
(after 6 '(1 2 3 4 3 5 6))     ⟹  (6)
(before 10 '(1 2 3 4 3 5 6))   ⟹  (1 2 3 4 5 6)
(before 1 '(1 2 3 4 3 5 6))    ⟹  (1)
(before 3 '(1 2 3 4 3 5 6))    ⟹  (1 2 3)
```
4.

Úkoly k textu

1.

2.

3.

Řešení ke cvičením

1.
```
(define abs-min2
  (lambda (x y)
    (if (<= (abs x) (abs y)) x y)))

(define abs-min
  (lambda args
    (foldr abs-min2 (car args) (cdr args))))
```
2.
```
(define nondecreasing?
  (lambda (l)
    (foldr (lambda (x y)
             (if y
                 (if (or (equal? y #t) (<= x y))
                     x
                     #f)
                 #f))
           #t
           l)))
```

3.
```
(define conseq
  (lambda (elem x y)
    (if (car y)
        y
        (cons (equal? x elem)
              (cons x (cdr y))))))
(define after
  (lambda (elem l)
    (let ((found (foldr (lambda (x y) (conseq elem x y)) '(#f . ()) l)))
      (if (car found)
          (cdr found)
          #f))))
(define before
  (lambda (elem l)
    (let ((found (foldl (lambda (x y) (conseq elem x y)) '(#f . ()) l)))
      (if (car found)
          (reverse (cdr found))
          #f))))
```