TEORIE PROGRAMOVACÍCH JAZYKŮ

Syntaktická a datová abstrakce

Petr Kaláb

1 SYNTAKTICKÁ A DATOVÁ ABSTRAKCE

V této kapitole uvedeme některé speciální konstrukce a příklady syntaktické abstrakce, které abstrahují společné syntaktické vzory. Neformálně jsou známy jako *syntaktický cukr*, protože dělají jazyk příjemnější pro použití. Nejprve uvedeme syntaktické abstrakce, které jsou užitečné k vytváření lokálních vazeb.

1.1 LOKÁLNÍ VAZBA

Definice vytváří vazbu na nevyšší úrovni oblasti programu. Často je ovšem potřeba vytvořit lokální vazbu. Uvedeme dvě speciální formy pro vytváření takovýchto vazeb.

1.1.1 LET

Mějme následující výraz:

```
(define subst
        (lambda (new old slst)
        …
                (if (symbol? (car slst))
                        (if (eq? (car slst) old)
                                (cons new (subst new old (cdr slst)))
                                (cons (car slst) (subst new old (cdr slst))))
                (cons (subst new old (car slst))
                        (subst new old (cdr slst))))
        …))
```

Výraz (car slst) se v předcházejícím příkladu vyskytuje čtyřikrát. Bylo by lepší vypočítat tento podvýraz pouze jednou a výslednou hodnotu navázat na proměnnou, na kterou se dále odkazovat. Protože tato vazba má význam pouze uvnitř tohoto výrazu, můžeme vytvořit lokální vazbu pouze pro tento výraz. Výhoda tohoto přístupu spočívá v zjednodušení výrazu a zefektivnění (výraz vyhodnocen pouze jednou).

Pro takovýto přístup používáme konstrukci *let*.

```
(let    ((var1 exp1)
        …
        (varn expn))
        body)
```

Oblast platnosti proměnných $var_1 \ldots var_n$ je v *body*. Každý výraz $exp_1 \ldots exp_n$ je vyhodnocen, proměnné $var_1 \ldots var_n$ jsou navázány na příslušné hodnoty a na závěr je vyhodnoceno tělo příkazu *let* a vrácena jeho hodnota. Tedy předchozí konstrukce je ekvivalentní konstrukci

```
((lambda (var1 … varn) body)
        exp1 … expn)
```

Občas potřebujeme dvě lokální vazby, kdy hodnota jedné závisí na hodnotě druhé. V takovém případě musí být použit vložený *let* výraz. Například:

(*let* ((x 3))

(*let* ((y (+ x 4)))

(* x y)))

což není ekvivalentní s

(*let* ((x 3)

(y (+ x 4)))

(* x y))

protože oblast platnosti proměnné *x* je v těle *let* příkazu. Nezahrnuje tedy definování *y* jako (+ x 4).

1.1.2 LETREC

Často je vhodné vázat procedury lokálně. Pokud procedura není použita v dalším průběhu programu, je dobrá zkušenost omezit její rozsah pouze na oblast kódu, kde je potřeba.

Zavedeme speciální formu *letrec*, která nám umožní lokální rekurzivní definice.

(*letrec* ((*var*$_1$ *exp*$_1$)

…

(*var*$_n$ *exp*$_n$))

body)

Definice je podobná s *let* až na to, že oblast deklarací *var*$_1$ … *var*$_n$ je celá *letrec* výrazem, včetně výrazů *exp*$_1$ … *exp*$_n$. Tedy *exp*$_1$ … *exp*$_n$ mohou definovat vzájemně rekurzivní procedury. Ve většině případů použití *letrec* jsou *exp*$_1$ … *exp*$_n$ lambda výrazy, ale není to podmínkou. Nicméně je však požadováno, že se nesmí provádět žádné odkazy na *var*$_1$ … *var*$_n$ během vyhodnocování *exp*$_1$ … *exp*$_n$. Například:

(*letrec* ((x 3)

(y (+ x 1)))

y)

je nelegální. Omezení je nezbytné, protože vazba na *var*$_1$ … *var*$_n$ nemusí mít žádnou hodnotu, dokud nejsou vyhodnoceny výrazy *exp*$_1$ … *exp*$_n$. Požadavek je jednodušší pokud tyto výrazy jsou lambda výrazy, protože odkazy na proměnné uvnitř těla lambda výrazu jsou vyhodnoceny pouze v případě, když je volána procedura, ne když je vyhodnocován lambda výraz. Tedy:

(*letrec* ((x 3)

(y (*lambda* () (+ x 1))))

y)

je legální a po vyhodnocení dává výsledek 4.

Je zde několik výhod proč používat *let* a *letrec* místo *define*.

- Při studiu volání procedur je hledání lokálních deklarací snazší než hledání deklarací globálních.
- Kód, který může být modifikován na proceduru, je omezen rozsahem lokálních deklarací.
- Redukce počtu globálních deklarací. Při použití globálních definic je vyšší pravděpodobnost výskytu konfliktu mezi deklarovanými proměnnými, kdy stejný název může být použit pro více než jednu deklaraci.

1.2 LOGICKÉ SPOJKY

Většina programovacích jazyků poskytuje způsob vyjádření *konjunkce* výrazů pomocí spojky *and*. Pro konjunkci platí, že nabývá true pokud jsou všechny její podvýrazy pravdivé. Podobně logická disjunkce může být vyjádřena spojkou *or* a nabývá true v případě, že alespoň jeden z podvýrazů disjunkce je pravdivý.

V některých programovacích jazycích jsou *and* a *or* reprezentovány procedurami. V takových případech jsou podvýrazy konjunkce a disjunkce vždy vyhodnoceny ještě před voláním samotné funkce. To může být zbytečné, protože pokud je v konjunkci nalezen podvýraz nepravdivý, kompletní výraz je také nepravdivý. Podobně u disjunkce, pokud je nalezen pravdivý podvýraz, potom celý výraz je pravdivý. V těchto případech nepotřebujeme vyhodnocovat všechny zbylé podvýrazy. Vyhodnocování všech podvýrazů v konjunkci a disjunkci je plýtvání výpočtem a také omezuje způsob použití konjunkce a disjunkce. Uvedeme příklad takového omezeni:

(*and* (*pair*? *x*) (*number*? (*car x*)))

Podvýraz (*car* x) může být vyhodnocen pouze v případě pokud je x dvojice. Pokud je *and* implementován jako procedura, vyhodnocují se oba dva podvýrazy. V případě, že x není dvojice, dojde k chybě.

V spoustě jazyků je konjunkce a disjunkce implementována ve speciálním tvaru vyhodnocování podvýrazů zleva doprava a nevyhodnocuje nezbytně všechny podvýrazy. Libovolná hodnota odlišná od #*f* (reprezentuje hodnotu *false*) je považována za pravdivou. Pokud je vypočtený výraz pravdivý, je užitečné vrátit hodnotu jako výsledek *and* a *or*. Tedy *and* vrátí hodnotu posledního podvýrazu a *or* vrátí hodnotu prvního pravdivého podvýrazu.

> (*and* #t (*number*? 3) 4)

4

> (or (number? #t) 3 4)

3

1.3 VĚTVĚNÍ

V této sekci uvedeme speciální formy *cond* a *case*, které vykonávají více-cestné větvení založené na vícenásobných testech.

1.3.1 COND

Speciální tvar *cond* slouží k více-cestnému větvení založeném na testování výrazů. Syntaxe je následující:

(*cond*

(*test*$_1$ *consequent*$_1$)

…

(*test*$_n$ *consequent*$_n$)

(*else alternative*))

Klíčové slovo *else* je součástí syntaxe *cond*. Výrazy jsou testovány postupně dokud některý z nich není pravdivý. Následně je vyhodnocen příslušný konsekvent a jeho hodnota je vrácena jako výsledek výrazu *cond*. Pokud není žádný z testovaných výrazů pravdivý, vrátí se vyhodnocený výraz *alternative* jako výsledek *cond*. Tedy *cond* je ekvivalentní sérii if příkazů.

Příkaz *cond* s jedním konsekventem je ekvivalentní jednoduchému if příkazu. Klauzule (*else alternative*) je nepovinná. Pokud chybí a žádný z testovaných výrazů není pravdivý, výsledná hodnota *cond* výrazu je nespecifikovaná.

1.3.2 CASE

Dovoluje porovnávat hodnotu s množinami symbolů nebo čísel. Syntaxe je následující:

(*case key*

(*key-list*$_1$ *consequent*$_1$)

…

(*key-list*$_n$ *consequent*$_n$)

(*else alternative*))

kde *key-list* je seznam symbolů, čísel, booleovských hodnot nebo písmen. Klausule *else* je nepovinná. Výraz *key* je vyhodnocen a jeho hodnota je porovnávána s elementy jednotlivých seznamů *key-list*. Následně je vyhodnocen konsekvent, který odpovídá prvnímu úspěšnému porovnání. Při neúspěšném porovnání se všemi seznamy *key-list* je jako výsledek příkazu *case* vrácen výsledek po vyhodnocení *alternative*, pokud se příslušná část v *case* příkazu vyskytuje, nebo je vrácena nespecifikovaná hodnota.

1.4 ZÁZNAMY

Doposud jsme používali pro složená data dvojice nebo vektory. Přístup k jednotlivým prvkům závisel na pozici v příslušné struktuře. Mnoho programovacích jazyků podporuje struktury nazývané záznamy, kde se k jednotlivým elementům přistupuje podle jména.

1.4.1 DEFINE-RECORD

Každý typ záznam má své jméno a množinu polí, kde každé pole má své jméno. Nový typ záznam můžeme definovat pouze na nejvyšší úrovni použitím *define-record*. Definice má tvar:

(*define-record name* (*field*$_1$ … *field*$_n$))

Tento zápis definuje proceduru pro vytvoření typu záznam s názvem *name*, predikát identifikující záznamy tohoto typu a přístupovou proceduru pro každé pole. Vytvářená procedura, nazývaná *make-name*, vezme *n* argumentů a vrátí nový záznam daného typu, kde *field*$_i$ obsahuje hodnotu i-tého argumentu. Typový predikát, nazývaný *name*?, vrací true pokud

je záznam daného typu. Přístupová procedura *name -> field*$_i$, pro $1 \leq i \leq n$, vrací ze záznamu hodnotu na i-té pozici.

Jako příklad můžeme uvést reprezentaci stromů. Mějme následující reprezentaci stromu:

<*tree*> ::= <*number*> | (<*symbol*> <*tree*> <*tree*>)

S výhodou můžeme použít záznam pro vnitřní uzly, které mohou reprezentovat symbol, levý podstrom a pravý podstrom. Situaci si demonstrujeme na příkladu:

> (*define-record interior* (*symbol left-tree right-tree*))

> (*define tree*-1 (*make-interior* '*foo* (*make-interior* '*bar* 1 2) 3))

> (*interior*? *tree*-1)

#t

> (*interior -> symbol tree*-1)

foo

> (*interior -> right-tree* (*interior -> left-tree tree*-1))

2

Definujme *leaf-sum*, která sečte hodnoty listů ve stromě:

```
(define leaf-sum
        (lambda (tree)
                (cond
                        ((number? tree) tree)
                        ((interior? tree) (+ (leaf-sum(interior -> left-tree tree))
                                             (leaf-sum(interior -> right-tree tree))))
                        (else (error "leaf-sum: Invalid tree" tree)))))
```

Dále definujme:

(*define-record leaf* (*number*))

1.4.2 PROMĚNNÉ ZÁZNAMY A VARIANT-CASE

Typ, který kombinuje dva typy nebo více typů jako alternativu je nazýván typ *union*. Strom, z předchozí kapitoly, je sjednocení záznamů typu *leaf* a *interior*. Typ *union* jehož všechny alternativy jsou záznamy se nazývá typ *variant rekord* (proměnný záznam). Mnoho programovacích jazyků poskytuje podporu pro typy *union* nebo *variant-record*. Syntaxe *variant-case*:

```
(variant-case record-expression
        (name_1 field-list_1 consequent_1)
        …
        (name_n field-list_n consequent_n)
        (else alternative))
```

kde pro $1 \leq i \leq n$ *field*$_i$ je seznam polí pro záznam typu *name*$_i$. Nejprve je vyhodnocen výraz *record*-expression, jeho hodnota je *v*. Pokud *v* není záznam určených typů, je vyhodnocen výraz *alternative* a vrácena jeho hodnota. Pokud je *v* záznam typu *name*$_i$, všechny jména polí ve *field-list*$_i$ jsou vázána na hodnoty polí, stejného jména, v záznamu *v*. Následně je vyhodnocen *consequent*$_i$ a vrácena jeho hodnota.

Předchozí definici *leaf-sum* můžeme zapsat následovně:

```
(define leaf-sum
    (lambda (tree)
        (variant-case tree
            (leaf (number) number)
            (interior (left-tree right-tree)
                (+ (leaf-sum left-tree) (leaf-sum right-tree)))
            (else (error "leaf-sum: Invalid tree" tree)))))
```

1.4.3 ABSTRAKTNÍ SYNTAX A JEHO REPREZENTACE S POUŽITÍM ZÁZNAMŮ

Programy, které zpracovávají další programy, jako například interprety nebo kompilátory, jsou obvykle *syntaxí řízené*. Co se provádí s jednotlivými částmi programu je dáno gramatickými pravidly.

Abstraktní syntax je taková reprezentace, že identifikuje syntaktická pravidla asociovaná s každou syntaktickou komponentou a obstará rychlý přístup k podkomponentám. Syntax, který je založen na lidské spotřebě je nazýván jako konkrétní syntax (concrete syntax).

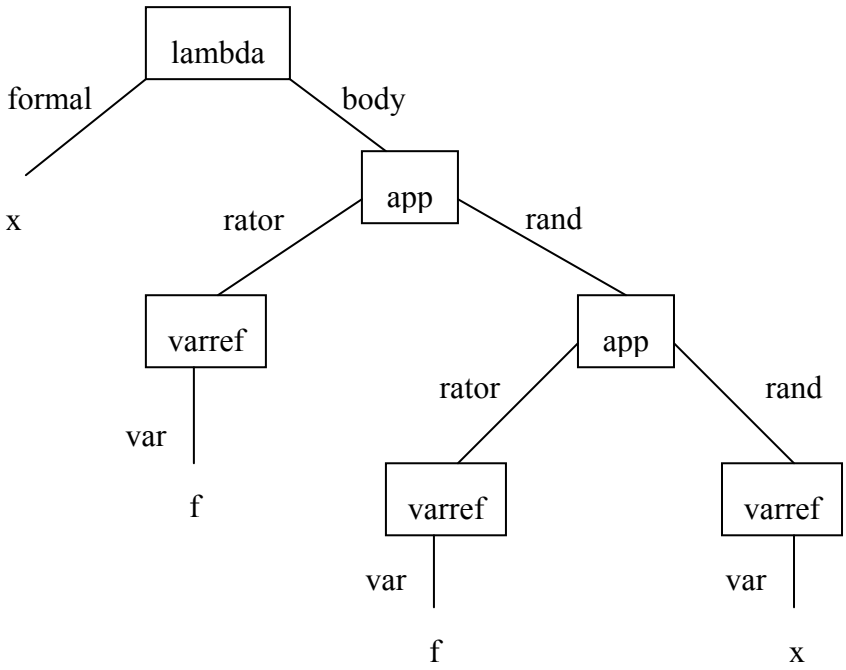
Při vytváření abstraktního syntaxu pro daný konkrétní syntax, musíme pojmenovat každé pravidlo konkrétního syntaxu a každý výskyt nonterminálu v každém pravidle. Jeden nonterminál se smí v jednom pravidle vyskytovat nejvýše jednou.

Mějme následující gramatiku:

<*exp*> ::= <*number*>	*lit* (*datum*)
| <*varref*>	*varref* (*var*)
| (*lambda* (<*var*>) <*exp*>)	*lambda* (*formal body*)
| (<*exp*> <*exp*>)	*app* (*rator rand*)

V pravém sloupci je uvedena syntaxe lambda kalkulu. Symboly *lit*, *varref*, *app*, *rator* a *rand* zkracují literál, odkaz na proměnnou (variable reference), aplikace, operátor a operand.

Reprezentace abstraktní syntaxe výrazu je nejrychleji vidět na abstraktním syntaktickém stromu. Pro příklad si uvedeme abstraktní syntaktický strom pro výraz v lambda kalkulu (*lambda* (*x*) (*f* (*f x*))).



Každý uzel ve stromu koresponduje s krokem v syntaktické derivaci výrazu. Ohraničené názvy znázorňují nonterminály, listy korespondují s terminálními symboly.

V programech je abstraktní syntax nejlépe reprezentován typy *variant record*. Typ *záznam* je asociován s každým jménem pravidla a jednotlivá pole (fields) jsou pojmenována podle odpovídajících nonterminálů, které se v pravidle vyskytují. Tedy pro lambda kalkul s čísly použijeme následující definice záznamů:

(*define-record lit* (*datum*))

(*define-record varref* (*var*))

(*define-record lambda* (*formal body*))

(*define-record app* (*rator rand*))

Pokud je program reprezentován řetězcem znaků, může být složitý problém derivovat odpovídající abstraktní syntaktický strom. Tato úloha, která se nazývá *parsing*, nesouvisí s tím co si přejeme provádět s abstraktním syntaktickým stromem. Takže činnost parsingu se nejlépe vykoná pomocí odděleného programu, nazývaného *parser*. Protože abstraktní syntaktické stromy jsou vytvářeny pomocí *parserů*, jsou také známy pod názvem *parse tree*.

1.4.4 IMPLEMENTACE ZÁZNAMŮ

Záznamy jsou abstraktní ve smyslu, že nemáme danou jejich přesnou reprezentaci. Mohou být reprezentovány jako vektory, seznamy nebo binární stromy založené na dvojicích nebo mohou být založené na úplně odlišné reprezentaci.

Uvedeme jednu možnou implementaci záznamů, v které jsou reprezentovány jako vektory. V mnoha případech je to nejúčinnější reprezentace ve smyslu časového a pamětové náročnosti. První element vektoru, který reprezentuje záznam, je symbol identifikující typ záznamu. Zbývající elementy obsahují pole hodnot v takovém pořadí, v kterém jsou nadefinovány pomocí příkazu *define-record*.

Jako příklad uvedeme definici listu u stromu:

(*define-record leaf* (*number*))

Tato definice produkuje definice ekvivalentní k:

```
(define make-leaf
        (lambda (number)
                (vector 'leaf number)))
(define leaf?
        (lambda (obj)
                (and (vector? obj)
                        (= (vector-length obj) 2)
                        (eq? (vector-ref obj 0) 'leaf))))
(define leaf->number
        (lambda (obj)
                (if (leaf? obj)
                        (vector-ref obj 1)
                        (error "leaf -> number: Invalid record" obj))))
```

1.5 DATOVÁ ABSTRAKCE

Datová reprezentace je často velmi složitá, proto se jí budeme vyhýbat. Například tabulka může být uchovávána jako vyvážený vyhledávací binární strom, ale my se nepotřebujeme starat o detaily údržby, vyhledávání a vyvažování stromu, kdykoliv volaná procedura přistupuje do tabulky nebo aktualizuje data v tabulce. Můžeme se také rozhodnou změnit reprezentaci dat. Nejefektivnější reprezentace je často nejobtížnější na implementaci. Můžeme si nejprve přát vytvořit jednoduchou implementaci a poté ji pouze měnit na účinnější reprezentaci pokud to bude mít rozhodující vliv na výkon celého systému. Například nevyvážené binární stromy, jednoduché seznamy mohou poskytovat adekvátní výkonovou reprezentaci tabulek a jsou daleko jednodušší a snazší na implementaci než vyvážené binární stromy. Často není předem známo, která reprezentace bude nejefektivnější dokud není získáno značné množství zkušeností s daným systémem, experimenty s alternativními reprezentacemi. Když se z nějakého důvodu rozhodneme změnit reprezentaci některých dat, musíme být schopni lokalizovat všechny úseky programu, které závisí na této reprezentaci.

Detaily reprezentace dat by měly být izolovány, protože potom nekomplikují porozumění programu a reprezentace dat může být změněna snadněji. Tento způsob se ustálil pro používání technik datové abstrakce. Nejprve se definují malé množiny procedur, které vytváří a operují nad daným typem dat a pouze tyto procedury přistupují k datům přímo. Tato množina procedur a vybraná datová reprezentace představují *abstraktní datový typ*, neboli ADT. Protože přístup k datům je možný pouze pomocí procedur ADT, zbytek programu je nezávislý na reprezentaci vybraných dat. Tato vlastnost se nazývá nezávislost reprezentace (representation independence). Tato vlastnost zajišťuje, že pokud je změněna reprezentace, ovlivní to pouze procedury ADT.

Vlastnosti ADT mohou být specifikovány ve způsobu nezávislosti reprezentace. Taková specifikace poskytuje rozhraní (interface) mezi ADT a zbytkem programu.

V některých systémech jsou typy disjunktní, každá hodnota je pouze jednoho typu. Mějme následující příklad:

> (*interior*? *tree*-1)

#t

> (*vector*? *tree*-1)

#t

> (*vector-ref tree*-1 0)

interior

Příklad demonstruje, že *tree*-1 je jak typu *interior* tak typu *vector*, tedy typy *vector* a *záznam* nejsou disjunktní. Dále příklad ukazuje, že záznamy jsou reprezentovány jako vektory, kde první element je název záznamu (viz. vector-ref).

Jestliže reprezentace typu je skrytá, nemůže být odhalena žádnou operací (včetně tisku), potom se takovému typu říká *opaque* (nepropustný?). Jinak řečeno je transparentní. Základními, primitivními typy jsou procedury, čísla, dvojice, vektory, znaky, symboly, řetězce, booleovský typ a prázdný seznam. Tyto typy jsou vzájemně disjunktní a opaque. Seznamy jsou odvozené typy skládající se z dvojic a prázdného seznamu. S typem seznam není prováděn žádný pokus o abstrakci, protože sdílí konstrukci a výběr procedur *cons*, *car* a *cdr* s typem dvojice (pair). Prázdný seznam je jediný typ s jedním prvkem, který je testován predikátem null?.

Chtěli bychom, aby typ záznam byl opaque, ale neexistuje standardní mechanismus jak vytvořit nový typ, který tuto vlastnost splňuje. (Schopnost definovat nový typ, který je opaque, je problematická. Opaque typy je těžké otestovat.)

1.6 OD PROCEDURÁLNÍCH K DATOVÝM STRUKTURÁM REPREZENTACE

Viděli jsme, že když je použitá datová abstrakce, programy mají vlastnosti reprezentační nezávislosti: Programy jsou nezávislé na částečné reprezentaci použité k implementaci ADT. Je potom možné změnit reprezentaci předefinováním malého počtu procedur náležících do ADT. Nejprve vyjádříme data jako procedury. Tato „high-level“ reprezentace výstižně přepracovává zamýšlenou sémantiku datových typů. (Další výhodou používání procedur k reprezentaci nových typů je, že reprezentace je svým způsobem nepropustná (opaque), protože procedury jsou sami o sobě obvykle nepropustné. Například, pokud ji vytiskneme, můžeme vidět, že to je procedura, ale nic víc.)

1.6.1 PROCEDURÁLNÍ REPREZENTACE

V této části ilustrujeme transformaci z procedurální do záznamové reprezentace použitím datového typu pro konečné funkce. Konečná funkce asociuje hodnotu s každým elementem konečné množiny symbolů. Je mnoho způsobů použití konečných funkcí. Jedním příkladem je prostředí, které asociuje proměnné s jejich hodnotami v implementaci programovacího jazyka.

Nejjednodušší způsob vytváření konečných funkcí je mít reprezentaci prázdné konečné funkce, která nevytváří žádné asociace, a mít způsob přidávání nových symbolů / hodnot existujícím konečným funkcím. To je základem rozhraní našich konečných funkcí ADT, které se skládají z následujících tří procedur:

1. *create-empty-ff* je procedura bez parametrů, která vrací prázdnou konečnou funkci, která nevytváří žádnou asociaci.
2. *extend-ff* vezme symbol, *sym*, hodnotu, *val*, a konečnou funkci, *ff*, a vrátí novou konečnou funkci, která uchovává asociace *ff* a také asociuje *sym* a *val*. Libovolná asociace se *sym*, která již může existovat v *ff* je novou funkcí ignorována.
3. *apply-ff* vezme konečnou funkci a symbol a vrátí hodnotu asociovanou se symbolem.

Výsledné rozhraní může být definováno následovně:

```
(define create-empty-ff
        (lambda ()
                (lambda (symbol)
                        (error "empty-ff: no association for symbol" symbol))))
(define extend-ff
        (lambda (sym val ff)
                (lambda (symbol)
                        (if eq? symbol sym)
                                val
                                (apply-ff ff symbol)))))
(define apply-ff
        (lambda (ff symbol)
                (ff symbol)))
```

Prázdná konečná funkce, vytvořená voláním *create-empty-ff*, signalizuje chybovou hláškou, že daný symbol není v její doméně. Procedura *extend-ff* vrací novou proceduru, která reprezentuje rozšířenou konečnou funkci. Na závěr procedura *apply-ff* jednoduše vykonává aplikaci.

1.6.2 ZÁZNAMOVÁ REPREZENTACE

Procedurální reprezentace je jednoduchá na pochopení, ale vyžaduje, aby byly procedury jako první třídy objektů. Můžeme systematicky transformovat tuto reprezentaci do jiné používající záznamy bez informace, která je obsažena v každé proceduře konečných funkcí. Jsou pouze dva druhy procedur konečných funkcí: jedny jsou získané procedurou pro vytvoření prázdné konečné funkce a další získané funkcí *extend-ff*. Specifikace toho co je vykonáno, když jsou tyto procedury volány je dána těly příslušných výrazů. Protože se tyto těla nemění, konečné funkce potřebují obsáhnout pouze příznak, kterou proceduru reprezentují. Následně se podle tohoto příznaku rozhodne, jaké příslušné akce mají být provedeny, když je volána *apply-ff*. Ale v (*lambda* (*symbol*) …) lambda výrazu *extend-ff* se vyskytují volné proměnné *sym*, *val* a *ff*. Procedura vytvořená vyhodnocením lambda výrazu musí nahrát hodnoty všech proměnných, které se vyskytují volně v těle lambda výrazu. Nahrané hodnoty musí být navázány ve chvíli, kdy je procedura vytvořena.

Uzavřeme to tak, že zachytíme informace, které jsou jedinečné pro každou proceduru konečné funkce do dvou záznamových typů:

(*define-record empty-ff* ())

(*define-record extend-ff* (*sym val ff*))

Můžeme implementovat abstrakci konečné funkce předefinováním *create-empty-ff* a *extend-ff* k vytvoření vhodných záznamů, předefinováním *apply-ff* k interpretaci informace v těchto záznamech a k vykonání akcí specifikovaných v těle příslušného (*lambda* (*symbol*) …) výrazu. Definice konečné funkce ADT s použitím této nové reprezentace je následující:

```
(define create-empty-ff
        (lambda ()
                (make-empty-ff)))
(define extend-ff
        (lambda (sym val ff)
                (make-extend-ff sym val ff)))
(define apply-ff
        (lambda (ff symbol)
                (variant-case ff
                        (empty-ff ()
                                (error "empty-ff: no association for symbol" symbol))
                        (extend-ff (sym val ff)
                                (if (eq? symbol sym)
                                        val
                                        (apply-ff ff symbol)))
                        (else (error "apply-ff: Invalid finite function" ff)))))
```

Sledujme, že konsekventy výrazů ve výrazu *variant-case* jsou přesně ty samé jako jsou těla příslušející (*lambda* (*symbol*) …) výrazům v procedurální reprezentaci. Také výše uvedená definice může být nahrazena:

(*define create-empty-ff make-empty-ff*)

(*define extend-ff make-extend-ff*)

Zajímavostí výše uváděné techniky transformace procedurální reprezentace na reprezentaci záznamovou je její obecnost. Může být aplikována na libovolnou množinu procedur, které reprezentují ADT. Dále zajišťuje, že tyto procedury jsou vždy volány aplikováním procedury, která odpovídá danému datovému typu. Klíčové kroky v transformaci jsou následující:

1. Identifikovat lambda výrazy, jejichž vyhodnocením dostáváme hodnoty typu, a vytvořit odlišný typ záznam pro každý z těchto výrazů.
2. Identifikovat volné proměnné v těchto lambda výrazech, jejichž hodnoty jsou různé pro každý element, a přidělit pole odpovídajícího typu záznamu pro každou z těchto proměnných.

3. Definovat aplikační proceduru pro typ používající výraz *variant-case* s jednou větví case na záznamový typ, kde seznam proměnných každého case seznamu volných proměnných identifikovaných v (2) a kde výraz konsekvent každého casu je tělo odpovídajícího lambda výrazu.

Předpokládejme, že si přejeme definovat proceduru extend-ff* která vezme seznam symbolů, sym-list, seznam hodnot, val-list, a konečnou funkci, ff, a vrátí novou konečnou funkci, která asociuje každý symbol ze sym-list s hodnotou na odpovídající pozici ve val-list.

```
(define extend-ff*
    (lambda (sym-list val-list ff)
        (if (null? sym-list)
            ff
            (extend-ff (car sym-list) (car val-list)
                (extend-ff* (cdr sym-list) (cdr (val-list) ff))))))
```

1.7 SHRNNUTÍ

Některé zajímavé vzory použití nejsou vyjádřeny jako procedury. Některé z nich jsou zapouzdřené jako syntaktické abstrakce. Zahrnují formy lokálních vazeb *let* a *letrec*, logické spojky *and* a *or* a formy více-cestného větvení *cond* a *case*. Syntaktické abstrakce *define-record* a *variant-case* poskytují rozhraní pro práci s proměnnými záznamy (variant-records). Abstraktní syntax reprezentuje syntaktickou strukturu výrazu jako strom.

Abstraktní datový typ, ADT, definuje rozhraní pro třídu dat. Je možné reprezentaci dat později změnit, stačí pouze předefinovat procedury, které definují rozhraní daného typu. Programy, které toto rozhraní používají, není potřeba měnit. To je princip datové abstrakce.

Často je vhodné nejprve reprezentovat data jako procedury. Takovou reprezentaci lze následně systematicky modifikovat na jinou, která je založena na záznamech, kde každý záznam obsahuje stejné dynamické informace, které jsou v odpovídajících procedurách.

2 REDUKCE PRAVIDEL A IMPERATIVNÍ PROGRAMOVÁNÍ

V této kapitole zavedeme transformační pravidla pro rozhodování o procedurách. Tyto pravidla jsou studována pro jednoduchost pouze v souvislosti s lambda kalkulem. Dále ukážeme vyhodnocovací mechanismus lambda kalkulu a jak používá transformační pravidla.

2.1 ÚVAHA O PROCEDURÁCH

Abychom zjistili výsledek volání procedury, musíme se podívat na její tělo. Pokud narazíme v těle procedury na odkaz na její formální parametr, můžeme si představit na daném místě proměnnou, která odpovídá argumentu z volání procedury. Například mějme definici:

```
(define foo
    (lambda (x y)
        (+ (* x 3) y)))
```

a procedury zavoláme jako:

(*foo* (+ 4 1) 7)

výsledek získáme vyhodnocením výrazu (+ (* (+ 4 1) 3) 7), což je 22.

Použili jsme pravidlo, které říká: Výsledek volání procedury může být získán nahrazením proměnných v těle procedury za odpovídající operandy, s kterými je procedura volána.

Tento postup můžeme vyjádřit jako:

(*foo* (+ 4 1) 7)

⇒ ((*lambda* (x y) (+ (* x 3) y))

(+ 4 1)

7)

⇒ (+ (* (+ 4 1) 3) 7)

⇒ 22

Druhá možnost je nejprve vyhodnotit argumenty a poté provést substituci.

(*foo* (+ 4 1) 7)

⇒ (*foo* 5 7)

⇒ (+ (* 5 3) 7)

⇒ (+ 15 7)

⇒ 22

Definujme:

(*define* c+

(*lambda* (*n*)

(*lambda* (*m*)

(+ *n m*))))

V minulé kapitole jsme viděli, že některé speciální formy, jako například *let*, jsou takovým syntaktickým cukrem pro ostatní formy. Taková pravidla můžeme vyjádřit například takovýmto způsobem:

(*let* ((*var*$_1$ *exp*$_1$) … (*var*$_n$ *exp*$_n$))

body)

⇒ ((*lambda* (*var*$_1$ … *var*$_n$) *body*)

exp$_1$ … *exp*$_n$)

Taková pravidla pak dále mohou být použita ve spojení s voláním procedur a následnému odvození složitějších výrazů.

(*let* ((*x* 3)

(*add5* (c+ 5)))

(*add5 x*))

⇒ (*let* ((*x* 3)

(*add5* (*lambda* (*m*) (+ 5 *m*))))

(*add5 x*))

⇒ ((*lambda* (*x add5*) (*add5 x*))

3

(*lambda* (*m*) (+ 5 *m*)))

⇒ ((*lambda* (*m*) (+ 5 *m*)) 3)

⇒ (+ 5 3)

⇒ 8

V předchozí diskusi jsme užívali několikrát termín „literálová reprezentace hodnoty“. Nemůžeme položit ve výrazu samotné hodnoty. Výše uvedený příklad obsahuje pouze numerické literály, proto může být obtížnější vidět rozdíl. Rozdíl je viditelnější, když hodnoty zahrnují také seznamy. Označíme literálovou reprezentaci seznamu pomocí apostrofu.

(*let* ((*second* (*lambda* (*x*) (*car* (*cdr x*)))))

(*second* (*list* 1 2 3)))

⇒ ((*lambda* (*second*) (*second* (*list* 1 2 3)))

(*lambda* (*x*) (*car* (*cdr x*))))

⇒ ((*lambda* (*x*) (*car* (*cdr x*)))

(*list* 1 2 3))

⇒ ((*lambda* (*x*) (*car* (*cdr x*)))

'(1 2 3))

⇒ (*car* (*cdr* '(1 2 3)))

⇒ (*car* '(2 3))

⇒ 2

Použitím apostrofu označujeme výskyt literálové reprezentace seznamu ve výrazu.

2.2 LAMBDA KALKUL A β-KONVERZE

Při studiu transformačních pravidel se omezíme pouze na množinu lambda kalkulu. Tento jazyk má pouze odkazy na proměnné, lambda výrazy s jedním formálním parametrem a volání procedur. Vše je dáno následující gramatikou:

<*exp*> ::= <*varref*>

| (*lambda* (<*var*>) (<*exp*>)

| (<*exp*> <*exp*>)

Často jsou přidávány do lambda kalkulu konstanty. Vytvářejí vhodnější nástroj pro vyjádření jednoduchých příkladů. Čísla používáme jako konstanty.

<*exp*> ::= <*number*>

V lambda kalkulu jsou transformační pravidla aplikována na volání ve tvaru:

((*lambda* (*var*) *exp*) *rand*)

Základní myšlenka je taková, že takovýto výraz je ekvivalentní výrazu získanému nahrazením odkazů *var* v sekci *exp* výrazem *rand*. Tuto situaci můžeme zapsat jako *exp*[*rand*/*var*]. Při přejmenovávání proměnných musíme dávat pozor, aby nedocházelo ke konfliktům. Může totiž nastat situace, kdy volná proměnná ve výrazu *rand* se stane po přejmenování vázanou proměnnou v lambda výrazu (vázaná přes *var*). Uvedeme si příklad nekorektní transformace:

((*lambda* (*x*)

(*lambda* (*y*) (*x y*)))

(*y w*))

transformujeme na

(*lambda* (*y*) ((*y w*) *y*))

Odkaz na *y* v (*y w*) zůstává volný, ale je zachycen vnitřním lambda výrazem.

Problém lze vyřešit změnou jména vnitřní proměnné *y* na proměnnou, která se nevyskytuje v argumentu (*y w*):

((*lambda* (*x*)

(*lambda* (*z*) (*x z*)))

(*y w*))

Takový výraz je ekvivalentní původnímu výrazu. Nyní můžeme vyhodnotit substituci následovně:

(*lambda* (*z*) ((*y w*) *z*))

Definujme substituci *M* pro *x* v *E*, psáno *E*[*M*/*x*]. Indukce je založena na tvaru E, s oddělenými pravidly pro každý tvar výrazu v lambda kalkulu. Pokud *E* je proměnná *x*, výsledek je *M*.

$x[M/x] = M$

Pokud je *E* jakákoliv jiná proměnná, *y*, nebo konstanta, *c*, výsledek je jednoduše tato proměnná nebo konstanta.

$y[M/x] = y$, kde $y \neq x$

$c[M/x] = c$

Pokud je *E* aplikace tvaru (*F G*), jednoduše vykonáme substituci na *F* a *G*.

$(F\ G)[M/x) = (F[M/x]\ G[M/x])$

Zajímavá situace nastane pokud výraz *E* má tvar: (*lambda* (*y*) *E*′). Pokud *y* je stejné jako *x* dostáváme:

(*lambda* (*x*) *E*′) [*M*/*x*] = (*lambda* (*x*) *E*′).

Pokud *x* není volné v *E*′.

(*lambda* (*y*) *E*′) [*M*/*x*] = (*lambda* (*y*) *E*′), kde *x* není volné v *E*′.

Pokud $y \neq x$ a *y* není volné v *M*, potom můžeme vykonat substituci na *E*′.

(*lambda* (*y*) *E′*)[*M*/*x*] = (*lambda* (*y*) *E′*[*M*/*x*]), kde *y* není volné v *M*.

Další případ, který jsme ještě neřešili je pokud $y \neq x$ a *x* je volné v *E′* a *y* je volné v *M*. Řešení je přejmenovat *y* (pomocí α-konverze) na nějaké jiné jméno, třeba *z*, které se nevyskytuje volně v *M* ani *E′*.

(*lambda* (*y*) *E′*)[*M*/*x*] = (*lambda* (*z*) (*E′*[*z*/*y*])[*M*/*x*]), kde z není volné v *E′* ani v *M*.

Tyto uvedené definice nám umožňují definovat volání procedur jako:

((*lambda* (*x*) *E*) *M*) = *E*[*M*/*x*]

a nazývá se β-*konverze*. Výraz ve tvaru ((*lambda* (*x*) *E*) *M*), na který může být aplikována β-*konverze* se nazývá β-*redex*.

Pokud je pravidlo β-*konverze* použito z leva do prava k transformování β-*redex*, nazývá se β-*redukce* a používá se k zjednodušení výrazů.

β-*konverze* může být použita také ve směru z prava do leva. To je to co je prováděno, když použijeme v programu příkaz let, abychom zabránili opakování některých výrazů. Například:

((*f* (*a* (*b c*))) (*a* (*b c*)))

⇒ ((*lambda* (*x*) ((*f x*) *x*))

(*a* (*b c*)))

⇒ (*let* ((*x* (*a* (*b c*))))

((*f x*) *x*))

2.3 REDUKČNÍ STRATEGIE

β-redukce může být použita k modelování efektivního vyhodnocování aplikací. Problém je, že lambda výraz může mít více než jeden redex, tedy může být více způsobů jak výraz redukovat. Zkoumejme následující příklad:

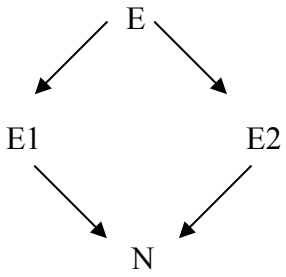
((*lambda* (*x y*) (+ (* *x* 3) *y*))

(+ 4 1)

7)

můžeme redukovat β-redex nebo výraz (+ 4 1). V této situaci to nemá vliv na výsledek a v obou případech dostáváme 22.

Vede nás to k otázce: Může být výraz redukován na více než jednu konstantu? Naštěstí ne. To ale není zřejmé pokud má výraz spoustu redexů a není nic řečeno o pořadí, v kterém byly redukce vykonávány. Existuje velmi zajímavý teorém, nazývaný *Church-Rosserův teorém*, který říká, že jestliže výraz E může být redukován na E1 nebo E2, použitím různých redukujících sekvencí, potom existuje nějaký výraz N, kterého můžeme dosáhnout z E1 a E2. Tento jev se nazývá Church-Rosserova, sbíhající nebo diamantová vlastnost. Můžeme jí vidět znázorněnou na následujícím obrázku.



Jak tento teorém odpovídá na naši otázku? Pokud je výraz redukován do dvou různých konstant, potom Church-Rosserův teorém říká, že existuje nějaký člen N do kterého mohu oba výrazy redukovat. Ale konstanty už dále redukovat nejdou. Z toho plyne, že nemohou existovat dvě různé konstanty, na které redukujeme týž výraz.

To je užitečné, ale říká nám Church-Rosserův teorém, že můžeme slepě aplikovat β-*redukci* a dojdeme k výsledku? Ne, protože ne každý výraz je redukovatelný na konstantu. Vezměme:

((*lambda* (*x*) (*x x*))

(*lambda* (*x*) (*x x*)))

Tento výraz je β-redex, ale β-redukcí dostáváme tentýž výraz.

Možná, že slabší platí výsledek: Říká nám Churh-Rosseův teorém, že pokud existuje řešení, jsme ho schopni nalézt slepým aplikováním β-redukce? Odpověď je opět záporná. Vezměme následující příklad:

((*lambda* (*y*) 3)

((*lambda* (*x*) (*x x*))

(*lambda* (*x*) (*x x*))))

Celý výraz je β-redex, který se okamžitě redukuje na 3, pokud *y* není volná proměnná v těle prvního lambda výrazu. Nicméně operand této aplikace je výraz, který jsme použili výše jako příklad výrazu, který nejde redukovat na konstantu. Proto tedy redukční sekvence, která opakovaně redukuje výraz operandu nikdy nenalezne řešení. Dvě odlišné strategie pro redukci výrazu se mohou lišit v tom zda naleznou řešení či nikoli.

2.3.1 APLICATIVE-ORDER REDUKCE

Další otázka zní: Jaký model redukční strategie je nejlepší? Je to strategie nazývaná applicative-order redukce. Pro applicative-order redukci je řešením buď konstanta, proměnná nebo aplikace (lambda výraz). V applicative-order redukci může být aplikována β-redukce pouze v takovém případě, že operand i operátor již mají řešení. Pokud řešení nemáme, potom může být zredukován operátor nebo operand.

Vytvořme proceduru *reduce-once-app1*, která vezme výraz v lambda kalkulu a redukuje jeho první *redex*, nalezne odpovídající applicative-order redukci, pokud takový redex existuje, a vrací úplný redukovaný výraz. Jestliže je výrazem konstanta, proměnná nebo lambda výraz, potom výraz nemůže být redukován. Jestliže je výraz aplikovatelný β-redex, jednoduše provedeme redukci a vrátíme výsledek. Pokud je výraz aplikace, ale není β-redexem, potom se nejprve pokusíme zredukovat operátor. Pokud neuspějeme pokusíme se redukovat operand. Kdykoliv jsme úspěšní s redukcí podvýrazu musíme zkonstruovat nový výraz, který obsahuje redukovanou formu podvýrazu.

Nějakým způsobem musíme indikovat zda volání *reduce-once-app1* proběhlo úspěšně či nikoli. Jedna možnost je vracet speciální návratovou hodnotu, která nám indikuje chybu. To je ale nešikovný způsob, protože po každém volání funkce *reduce-once-app1* musíme nahrát návratovou hodnotu pomocí *let* a poté ji testovat. Výkonnost závisí na počtu redukcí. Použijme tedy jiný přistup. Projdeme proceduru *reduce-once-app1* s dvěma přidanými argumenty: procedurou *succeed*, která se vykoná, když redukce výrazu proběhla úspěšně, a procedurou *fail*, která je volána v případě, že redukce selže.

Specifikace *reduce-once-app1* je

(*reduce-once-app1 exp succeed fail*) =

(*succeed exp′*) pokud *exp* obsahuje aplikovatelný β-redex a exp′ je výsledek po vykonání jedné applicative-order redukce na exp nebo

(*fail*) pokud exp nemá žádný aplikovatelný β-redex.

2.3.2 NEJLEVĚJŠÍ REDUKCE

Applicative-order redukce je dobrý model schématu popisujícího chování procedur. Nicméně jsme viděli, že tato metoda může selhat při hledání řešení, přestože nějaké existuje. Například applicative-order redukce selže na následujícím výrazu:

((*lambda* (*y*) 3)

((*lambda* (*x*) (*x x*))

(*lambda* (*x*) (*x x*))))

protože bude chycena v nekonečné smyčce pokoušející se zredukovat operand. Další otázkou je: „Existuje redukční strategie, která nám garantuje nalezení řešení, pokud existuje?“ Ano a nazývá se *nejlevější redukce* a je to strategie redukování β-redexu jehož levá závorka je jako první v pořadí.

Řekněme, že lambda výraz je v *normální formě*, pokud neobsahuje žádné β-redexy. Pokud je lambda výraz v normální formě, potom již nejsou možné žádné další redukce. Například lambda výraz (*lambda* (*x*) *x*) je v normální formě, ale není to konstanta. Church-Rosserův teorém nás vede k závěru, že výraz může mít nejvíce jednu normální formu. Může být ukázáno, že nejlevější redukce vždy nalezne normální formu výrazu, pokud taková forma existuje. Z tohoto důvodu se nejlevější redukce občas nazývá jako normal order redukce (redukce v normálním pořadí).

Nejlevější redukce je obecná, nicméně za cenu výkonnosti. Například vezměme následující sekvenci nejlevějších redukcí, kde operand ((*lambda* (*w*) *w*) *z*) je redukován dvakrát:

((*lambda* (*x*) (*x* (*x y*)))

((*lambda* (*w*) *w*) *z*))

⇒ (((*lambda* (*w*) *w*) *z*)

(((*lambda* (*w*) *w*) *z*)

y))

⇒ (*z* (((*lambda* (*w*) *w*) *z*) *y*))

⇒ (*z* (*z y*))

Pokud je operand ((*lambda* (*w*) *w*) *z*) redukován před voláním procedury (*lambda* (*x*) (*x* (*x y*))) stačí o jednu redukci méně, aby bylo dosaženo normální formy.

((*lambda* (*x*) (*x* (*x y*)))

((*lambda* (*w*) *w*) *z*))

⇒ ((*lambda* (*x*) (*x* (*x y*)))

z)

⇒ (*z* (*z y*))

Obecně je applicative-order redukce efektivnější než nejlevější redukce. Proto moderní programovací jazyky používají některé varianty založené na applicative-order redukci. Existuje několik programovacích jazyků používajících nejlevější redukci ke garantování toho, že vždy bude nalezena normální forma, pokud existuje. Překladače těchto jazyků provádějí typicky rozsáhlou analýzu, nazývanou přísná analýza (strictness analysis) k nalezení takových míst, kde je bezpečné použít applicative-order redukci.

2.4 DEFINOVÁNÍ REKURZIVNÍCH PROCEDUR V LAMBDA KALKULU

Zakončíme naší diskusi o lambda kalkulu problémem vyjádření rekurzivních procedur. Lambda kalkul neobsahuje *letrec* nebo *define*. Jak tedy můžeme definovat rekurzivní procedury v lambda kalkulu?

Předpokládejme, že *exp* je lambda výraz definující rekurzivní proceduru, kde rekurze je vykonaná odkazem na proměnnou, *g*, která je volná ve výrazu *exp*. Tedy *g* je jméno stejné procedury jako *exp*. Proměnná *g* musí být sama uvedena v lambda výrazu, jako

(*lambda* (*g*) *exp*)

Například pro definování procedury, která počítá faktoriál, můžeme psát něco takovéhoto:

(*lambda* (*g*)

(*lambda* (*n*)

(*if zero*? *n*) 1 (* *n* (*g* (- *n* 1))))))

kde předpokládáme, že lambda kalkul je rozšířený o *if*, *zero*?, *, - a 1 a *g* je myšlená funkce faktoriálu. Zavolejme tento výraz *f*. Jak může být tento výraz změněn, aby vyjadřoval rekurzivní funkci faktoriál?

Potřebujeme napsat proceduru, nazvanou Y takovou, že (Y f) je požadovaná rekurzivní procedura. Předpoklad, že rekurzivní funkce může být obdržena odkazem na *g*, může být uspokojen navázáním *g* na (Y f) s aplikací (f (Y f)). Platí následující rovnost:

(*Y f*) = (*f* (*Y f*))

Může být Y vyjádřen jako lambda výraz nebo musí být přidán do lambda kalkulu jako nějaká speciální vlastnost? Překvapivě Y může být definováno jako:

(*lambda* (*f*)

(*lambda* (*x*) (*f* (*x x*)))

(*lambda* (*x*) (*f* (*x x*)))))

Y se nazývá operátor pevného bodu (*combinator*). Poznamenejme, že operátor pevného bodu nemá normální formu.

Tato verze Y je nepoužitelná pokud používáme applicative-order redukci. Operátor pevného bodu pro applicative-order redukci:

(*lambda* (*f*)

((*lambda* (*x*) (*f* (*lambda* (*y*) ((*x x*) *y*))))

((*lambda* (*x*) (*f* (*lambda* (*y*) ((*x x*) *y*))))))

2.5 SHRNUTÍ

β-konverze v lambda kalkulu je mocný nástroj pro rozhodování o funkčnosti programu. Opakovaným aplikováním přepisuje pravidla pouze v jednom směru. Výrazy v lambda kalkulu mohou být redukovány až k řešení. Church-Rosserův teorém nám říká, že každý výraz může být redukován na nejvýše jednu konstantu. Pořadí, v kterém jsou vykonávány redukce, může ovlivňovat ukončení redukčního procesu. Nejlevější redukce zastaví kdykoliv je to možné, kdežto applicative-order redukce je obvykle efektivnější a je použita ve většině programovacích jazycích. Rekurzivní procedury mohou být definovány přímo v lambda kalkulu.

3 POUŽITÁ LITERATURA

[1] Friedman, D. P.; Wand, M.; Haynes, Ch. T.: Essentials of Programming languages. McGraw-Hill, 1992. ISBN 00-7022-443-9