Technická univerzita v Košiciach

Fakulta elektrotechniky a informatiky

Katedra počítačov a informatiky

Formálne špecifikácie úloh pre programovanie

Diplomová práca

Študijný odbor: Výpočtová technika a informatika

Vedúci diplomovej práce:

RNDr. Valerie Novitzká, PhD.

Diplomant:

Tomáš Klema

Konzultant diplomovej práce:

RNDr. Valerie Novitzká, PhD.

Košice 2004

Čestné vyhlásenie

Vyhlasujem, že som diplomovú prácu vypracoval samostatne s využitím uvedenej odbornej literatúry.

Košice 7. 5. 2004 . . . . . .

Vlastnoručný podpis

Poďakovanie

Považujem za svoju milú povinnosť poďakovať pani RNDr. Valerii Novitzkej, PhD., za odbornú pomoc a organizačné vedenie pri spracovaní tejto práce. Ďalej chcem poďakovať taktiež priateľom za cenné a vecné pripomienky a rady pri spracovávaní tejto práce. V neposlednom rade chcem poďakovať rodine za podporu a možnosť štúdia na vysokej škole.

Anotácia

Názov práce: Formálne špecifikácie úloh pre programovanie

Pracovisko: Katedra počítačov a informatiky, FEI TU Košice
Autor: Tomáš Klema
Vedúci DP: RNDr. Valerie Novitzká, PhD.
Konzultant DP: RNDr. Valerie Novitzká, PhD.
Dátum: 7. 5. 2004
Kľúčové slová: CASL, Haskell, zobrazenie, design, požiadavky

Anotácia: Prezentovaná práca sa zaoberá špecifikáciou požiadaviek a špecifikáciou návrhu – problémom korektného programovania. Problém transformácie je taktiež významným prvkom práce. Mnoho štúdií bolo venovaných formálnym špecifikáciám a malo by byť poznamenané, že tento proces je ešte stále vo vývoji. Rôzne mechanizmy transformácií môžu byť zvolené na vytvorenie inštitučnej šipky. Bol predložený koncept transformácie. Výsledok podaný v tejto práci môže byť použitý pri podpore procesu korektného programovania.

Anotation

Thesis title:	Formal specification of tasks for programming
Department:	Department of Computers and Informatics, TU FEI Košice
Author:	Tomáš Klema
Supervisor:	RNDr. Valerie Novitzká, PhD.
Tutor:	RNDr. Valerie Novitzká, PhD.
Date:	7. 5. 2004
Keywords:	CASL, Haskell, transformation, design, requirements

Annotation: Presented thesis deals with the requirements specifications and specification of design – problem of correct programming. The problem of transformation is also a great interest of the thesis. Many studies have already been devoted to the formal specification and it should be noted that this is still a process in development. Different transformation mechanisms may be chosen for creating an institution arrow. The concept of the transformation was introduced. Results of the submitted thesis can be applied in support of correct programming process.

Obsah

Úvod **1**

1 Formulácia úlohy **2**

2 Základy teórie **3**

2.1 Signatúra 3

2.2 Model 4

2.3 Inštitúcia 4

3 Jazykové prostriedky **8**

3.1 CASL – CoFI Algebraic Specification Language 8

3.1.1 Základná špecifikácia 9

3.1.2 Štrukturovaná špecifikácia 10

3.2 Syntax jazyka CASL 12

3.2.1 Základné konštrukcie 12

3.3 Haskell – funkcionálny jazyk 21

3.3.1 Typy 21

3.3.2 Funkcie 22

3.3.3 Typové triedy a preťaženie 23

4 Zobrazenie **24**

4.1 Konštrukcia špecifikácie požiadaviek 25

4.2 Konštrukcia špecifikácie návrhu 27

4.2.1 Signatúra 27

4.2.2 Formula 30

4.2.3 Model 31

4.3 Prechod od špecifikácie požiadaviek k špecifikácii návrhu 33

5 Príklady **35**

6 Zhodnotenie riešenia **40**

Zoznam použitej literatúry **41**

Zoznam príloh **42**

Zoznam obrázkov **43**

Zoznam tabuliek **44**

Úvod

Bola polnoc — čas zlý, krutý — a ja slabý, pochudnutý
študoval som staré spisy v hrubých zväzkoch zviazané.

E. A. Poe – Havran

Táto práca je jednou časťou z väčšieho celku – korektného programovania. Pod pojmom *korektný program* rozumieme taký spoľahlivý program, o ktorom sa dá matematicky dokázať, že po vykonaní spĺňa špecifikované požiadavky. V procese programovania by mal programátor najprv formulovať otázku (otázky), na ktorú chce získať odpoveď. Problém, na ktorý sa pýtame, by mal byť v danej oblasti riešiteľný, keďže snahou programátora je úrychliť riešenie problému pomocou počítača. Na základe týchto otázok sa formulujú predpoklady, ktoré musia byť splnené. Tieto predpoklady nazývame *špecifikáciou požiadaviek*. Táto špecifikácia už obsahuje matematicky vyjadriteľné tvrdenia. Tieto tvrdenia sa potom programátor snaží prepísať tak, aby získal čiastočnú odpoveď na svoju otázku, čím sa dostávame k *špecifikácii návrhu*[1]. V ďalších krokoch pokračuje proces vývoja softvéru k špecifikácii vykonávateľného programu a nakoniec k výslednej špecifikácii. Špecifikácia požiadaviek musí byť formulovaná v určitej matematickej teórii a môžme ju definovať ako:

- popis zobrazenia vstupnej údajovej štruktúry na výstupnú, alebo
- program – údajové štruktúry + algoritmy

Cieľom tejto práce je prezentovať špecifikačný jazyk a programovací jazyk, ich vlastnosti, výhody a stručný popis. Ďalej taktiež popis zobrazenia z jedného jazyka do druhého, pričom zobrazenie má byť matematicky podchytené a dokázateľné.

[1] v niektorých literatúrach označovanú ako *špecifikácia programového textu*

1 Formulácia úlohy

Zadanie tejto práce pozostáva z nasledujúcich bodov:

1. Definovať zobrazenie typového systému jazyka CASL na typový systém jazyka HASKELL.
2. Demonštrovať definované zobrazenie na príkladoch.
3. Zhodnotiť možnosti použitého prístupu a stanoviť problémy pre ďalší výskum v tejto oblasti.

Práca pozostáva zo šiestich kapitol. V úvode sú zhrnuté motivácia a ciele tejto diplomovej práce. Druhá kapitola stručne popisuje základné teoretické pojmy, ktoré sa vyskytujú v celej práci. Kapitola 3 popisuje jazykové sysémy použité v tejto práci, konkrétne jazyky CASL a HASKELL, ich syntax a základné vlastnosti. V kapitole 4 je rozobratá metodológia zobrazenia, konkrétne vytváranie inštitúcií. V piatej kapitole je uvedených niekoľko príkladov demonštrujúcich zobrazenie. A nakoniec posledná kapitola popisuje súhrn práce a zistené závery.

2 Základy teórie

V tejto kapitole si vysvetlíme základne pojmy, s ktorými budeme neskôr pracovať. Podrobnejšie rozpracované pojmy (funktor, kategória, inštitúcia) sú v [5].

2.1 Signatúra

Viacdruhová(many-sorted) signatúra $\Sigma = (S, TF, PF, P)$ pozostáva z:

- množiny druhov S;
- množín $TF_{w,s}$, $PF_{w,s}$, *úplných funkčných* resp. *čiastočných funkčných symbol*ov, za predpokladu, že $TF_{w,s} \cap PF_{w,s} = \emptyset$, pre každý *funkčný profil* (w, s) pozostávajúc zo sekvencie argumentov druhov $w \in S^*$ a *výsledného druhu* $s \in S$ (konštanty sú ošetrené ako funkcie bez argumentov);
- množiny P_w *predikátových* symbolov, pre každý *predikátový* profil pozostávajúci zo sekvencie argumentov druhov $w \in S^*$.

Symboly môžu byť preťažené, vyskytujúc sa vo viac ako jednej z uvedených množín. Na zabezpečenie jednoznačnosti vo vetách na tejto úrovni, symboly sú stále kvalifikované profilom pri použití.

Morfizmus viacdruhovej signatúry

$$\sigma : (S, TF, PF, P) \rightarrow (S', TF', PF', P')$$

pozostáva zo zobrazenia S na S' a pre každé $w \in S^*$, $s \in S$, zobrazenie medzi korešpondujúcimi množinami funkcií, resp. predikátovými symbolmi. Čiastočné funkčné symboly môžu byť zobrazené taktiež na úplné funkčné symboly ale nie naopak.

2.2 Model

Pre viacdruhovú signatúru $\Sigma = (S, TF, PF, P)$ je viacdruhový *model* $M \in \mathbf{Mod}(\Sigma)$ viacdruhovou štruktúrou prvého rádu pozostávajúc z viacdruhovej čiastočnej algebry:

- **neprázdny** nosič (carrier set) s^M pre každý druh $s \in S$ (nech w^M znamená kartézsky súčin $s_1^M \times ... \times s_n^M$ keď $w = s_1...s_n$),
- čiastočná funkcia f^M z w^M do s^M pre každý funkčný symbol $f \in TF_{w,s}$ alebo $f \in PF_{w,s}$, funkcia by mala byť úplná v prvom prípade,

spolu s:

- *predikátom* $p^M \subseteq w^M$ pre každý predikátový symbol $p \in P_w$.

Viacdruhový (slabý) homomorfizmus h z M_1 do M_2, s $M_1, M_2 \in Mod(S, TF, PF, P)$, pozostáva z funkcie $h_s : s^{M_1} \rightarrow s^{M_2}$ pre každý $s \in S$ zachovávajúc nielen jej hodnotu, ale aj jej definovanosť a zachováva pravdivosť predikátov.
Každý morfizmus signatúr

$$\sigma : \Sigma \rightarrow \Sigma'$$

určuje viacdruhovú redukciu každého z modelu $M' \in \mathbf{Mod}(\Sigma')$ na model $M \in \mathbf{Mod}(\Sigma)$, definovanú ako interpretáciu symbolov zo Σ v M v rovnakom smere ako ich obrazy pod σ sú interpretované v M'.[7]

2.3 Inštitúcia

V predchádzajúcich častiach sme zaviedli niektoré základné pojmy, ktoré teória programovania bude používať pri dokazovaní rôznych užitočných vlastností rôznych špecifikácií počas vývoja programu, vrátane korektnosti. Teraz uvedieme pomocnú definíciu, ktorá zoskupuje pripravené pojmy tak, že umožní

- formulovať nielen špecifikáciu, ale aj jej dokázané vlastnosti;
- jednoznačným spôsobom pojednávať o vývojovom grafe programov od špecifikácie požiadaviek až ku výslednej špecifikácií. Tento pomocný pojem nazývame *inštitúcia*.

Definícia 2.1 Nech **Sign** je kategória signatúr danej špecifikácie požiadaviek. Definujeme funktor Sen

$$Sen : \mathbf{Sign} \to \mathbf{Set}$$

z kategórie signatúr **Sign** do kategórie množín **Set**. Tento funktor

- každej signatúre Σ priradí množinu pravdivých uzavretých Σ–formúl $Sen(\Sigma) \subseteq \mathbf{Set}_{\mathbf{Obj}}$;
- každému morfizmu signatúr $\sigma : \Sigma \to \Sigma'$ priradí zobrazenie

  $$Sen(\sigma) : Sen(\Sigma) \to Sen(\Sigma')$$

  ktoré nahradí všetky symboly Σ–formúl ich obrazmi podľa σ v Σ'.

□

Teraz signatúre Σ priradíme kategóriu $\mathbf{Mod}(\Sigma)$. Objekty kategórie $\mathbf{Mod}(\Sigma)$ sú Σ–modely, ktoré pozostávajú z množiny, spolu s interpretáciou každého n–árneho funkčného symbolu ako n–árnej operácie a každého n–árneho predikátového symbolu ako n–árneho predikátu. Morfizmy tejto kategórie sú funkcie, ktoré zachovávajú operáciu a predikáty. Ak máme teda Σ–model M a Σ–vetu φ, môžeme medzi nimi utvoriť už spominaný vzťah splniteľnosti vety φ v modely M, čo zapisujeme ako $M \models_\Sigma \varphi$.

Definícia 2.2 Nech Σ je signatúra z kategórie **Sign**. Trieda Σ–modelov spolu so Σ–homomorfizmami medzi nimi tvorí *kategóriu Σ–modelov*

$$\mathbf{Mod}(\Sigma)$$

Definujeme redukujúci funktor $\lrcorner$ z kategórie Σ'–modelov do kategórie Σ–modelov

$$\lrcorner : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$$

ktorý

- Σ'–modelu A' priradí jeho σ–redukt, Σ–model $A'_{|\sigma}$;
- Σ'–homomorfizmu priradí Σ–homomorfizmus medzi odpovedajúcimi reduktami.

Zavedieme funktor Mod, ktorý zobrazí duálnu kategóriu signatúr do kategórie malých kaegórií

$$Mod : \mathbf{Sign^{op}} \to \mathbf{Cat}$$

pričom

- každej signatúre Σ priradí kategóriu Σ–modelov;
- každému morfizmu signatúr $\sigma : \Sigma \to \Sigma'$ priradí funktor z kategórie Σ'–modelov do kategórie Σ–modelov

$$Mod(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma).$$

□

$$\begin{tikzcd}
\Sigma \arrow[r, "\sigma"] & \Sigma' \\
M_{|\sigma} & M' \arrow[l] \\
\models_{\Sigma} & \models_{\Sigma'} \\
\psi_{\Sigma} \arrow[r] & \mathbf{Sen}(\sigma)(\psi_{\Sigma})
\end{tikzcd}$$

Obrázok 2 – 1 relácia splnenia formúl

Pomocou takto definovaných pojmov teraz definujeme inštitúciu[6].

Definícia 2.3 Inštitúcia **I** je usporiadaná štvorica

$$\mathbf{I} = (\mathbf{Sign}, Sen, Mod, (\models_\Sigma)_{\Sigma \in \mathbf{Sign}})$$

kde **Sign** je kategória signatúr špecifikácie, Sen a Mod sú funktory definované vyššie a

$$(\models_\Sigma)_{\Sigma \in \mathbf{Sign}}$$

je trieda relácií splnenia Σ–výrokoch v Σ–modeloch pre signatúry Σ z kategórie **Sign** viď obrázok 2 – 1.

□

3 Jazykové prostriedky

V tejto kapitole načrtneme špecifikačný a implementačný jazyk, medzi ktorými bude definované zobrazenie. Popíšeme základnú charakteristiku jazykov, ako aj ich syntax a sémantiku. Ako špecifikačný jazyk sme zvolili CASL (CoFI Algebraic Specification Language)[2], ktorý navrhla a ďalej vyvíja európska skupina CoFI (Common Framewrok Initiative). Za implementačný jazyk sme zvolili jazyk Haskell, kôli jeho funkcionálnemu základu a silnej typovosti.

3.1 CASL – CoFI Algebraic Specification Language

CASL je rozumne expresívny jazyk na špecifikovanie (požiadaviek, návrhu a architektúry) konvenčného softvéru. Slovo "algebraický" je použité v tomto dokumente vzhľadom na všeobecnú axiomatickú (vlastnosťami orientovanú) špecifikáciu tried modelov. Žiadne reštrikcie nie sú zahrnuté voči ekvacionálnej logike (equational logic). Aj keď prístup známy ako "algebraická špecifikácia" vychádzal len z algebraických metód, ktoré majú hlavne dobrú oblasť vlastností a konštrukcií (napr. počiatočné modely) a ktoré sú stále užitočné, môže byť výhodou povoliť celú logiku prvého rádu a indukčné princípy (druhého rádu) v záujme stručnosti a jasnosti.

Základná špecifikácia v CASL označuje triedy s čiastočnými štruktúrami prvého rádu: algebry, kde funkcie sú čiastočné alebo úplné a kde sú taktiež povolené predikáty. Poddruhy (subsorts) sú vkladané. Axiómy sú formuly prvého rádu postavené z definovateľného výroku a silnej a existenčnej rovnosti. Obmedzenia pri tvorbe druhov (sort) môžu byť určené. Deklarácie dátových typov sú poskytované pre stručnú špecifikáciu druhov vybavených s nejakými konštruktormi a (nepovinným) selektorom, vrátane enumerácií a produktov. Štrukturovaná špecifikácia dovoľuje zobrazenie, redukciu, zjednotenie a rozšírenie špecifikácie. Rozšírenia môžu byť požadované aby boli konzervatívne a/alebo voľné;

[2] **C** alternatívne môže znamenať Common a **A** Axiomatic

počiatočné ohraničenia sú špeciálnym prípadom. Jednoduchá forma všeobecnej (parametrizovanej) špecifikácie je poskytnutá spolu s inštanciovaním zahrňujúc preklad vhodnosti parametrov. Architektonická špecifikácia vyjadruje to, že špecifikovaný softvér je zložený z oddelene vyvynutých, znovapoužiteľných častí s jasným rozhraním. Nakoniec, knižnice dovoľujú (distribuované) uloženie a získanie pomenovaných špecifikácií.[8]

3.1.1 Základná špecifikácia

čiastočnosť (partiality) – táto vlastnosť je zo štandardnej matematickej interpretácie čiastočnosti: funkcie sú "striktné" s nedefinovateľnosťou ktoréhokoľvek parametra v aplikácii vynucujúce nedefinovateľnosť výsledku.

poddruh a preťaženie (subsorts and overloading) – bolo zistené, že je veľmi žiadúce poskytnúť podporu pre koncept poddruhov a preťaženia (ako napr. povoliť špecifikáciu prirodzených čísel ako poddruh celých čísel s klasickými funkciami nad množinou prirodzených čísel, rozšírenú na množinu celých čísel), ale myšlienka "regulárnosti" signatúr, prevzatá z postupne usporiadanej (order–sorted) algebry, má určité nedostatky.

formuly – používame klasické kvantifikátory prvého rádu a logické spojenia. Predikáty použité v atomických formulách možu byť tiež deklarované.

deklarácia premenných (variable declarations) – axiómy v algebraických špecifikáciách sú typicky určené aby boli univerzálne kvantifikované nad každou (usporiadanou) premennou, ktorá sa v nich objaví. Takéto kvantifikovanie môže byť explicitne v CASL-i s premennými určitých druhov deklarovaných pre použitie v individuálnych formulách.

viditeľnosť a platnosť (visibility and scope) – poradie v ktorom sú prvky základnej špecifikácie uvedené je nepodstatné. V mnohých prípadoch je ale

možné zoskupiť niektoré prvky v prirodzenom poradí. Za zmienku stojí, že štruktúrovaná špecifikácia CASL-u dovoľuje vyjadriť lineárnu viditeľnosť.

atribúty (attributions) – vlastnosti také ako asociatívnosť môžu byť priradené funkciám. Asociatívnosť, komutatívnosť, idempotentnosť a vlastnosti celku môžu byť taktiež dôležitou časťou nástrojov na prepisovanie termov. Atribúty sú rozdielne od axiómov.

obmedzenia pri tvrobe druhov (sort generation constraints) – vo všeobecnosti, CASL neobmedzuje všetky modely, aby boli konečno generované (finitely–generated), tj. dosiahnuteľné. Je ponechané na navrhovateľovi špecifikácie, aby vyjadril obmedzenia pri tvorbe druhov kde je to vhodné.

deklarácia dátových typov (datatype declaration) – deklarácia dátových typov povoľuje stručnú deklaráciu druhov spolu s konštruktormi a selektormi funkcií, s niekoľkými implicitnými axiómami súvisejúcimi s konštruktormi a selektormi. V praktickom špecifikačnom jazyku je dôležité, aby bolo možné vyhnúť sa zdĺhavým, opakujúcim sa schémam (patterns) špecifikácie, keďže sú pravdepodobne písané bez záujmu a presne nečítane.

3.1.2 Štrukturovaná špecifikácia

zobrazenie a utajenie (translation and hiding) – symboly deklarované špecifikáciou môžu byť zobrazené na rôzne symboly a tie môžu byť ukryté. Zobrazenie je primárne potrebné na povolenie znovupoužitia špecifikácie so zmenou notácie, ktorá je dôležitá keďže rôzne aplikácie môžu vyžadovať použitie rôznej notácie pre rovnaké entity. Ukrývanie symbolov zaručuje, že tieto nie sú dostupné pre používateľa špecifikácie, co je vhodné pre symboly označujúce pomocné entity zavedené návrhárom špecifikácie. To umožňuje úplne uľahčiť špecifikáciu a určiť, ktoré symboly nepotrebujú byť implementované.

zjednotenie a rozšírenie (union and extension) – špecifikácia nezávislých prvkov môže byť kombinovaná a následne rozšírená o špecifikáciu ďalších druhov, vložených poddruhov, funkcií, predikátov a/alebo vlastností. Zjednotenie sa využíva pri spájaní dvoch nezávislých špecifikácií, kde model poskytuje všetky symboly oboch špecifikácií. Rozšírenie špecifikácie dovoľuje pridávanie ďalších funkcií (a predikátov) k už špecifikovaným druhom.

počiatočnosť a voľnosť (initiality and freeness) – špecifikácie vo všeobecnosti majú voľnú sémantiku: všetky modely deklarovaných symbolov, ktoré využívajú špecifikovanú vlastnosť, sú povolené. Akokoľvek, môže byť taktiež špecifikované, že len počiatočné modely špecifikácie sú povolené. Špecifikácie s voľnou sémantikou a počiatočnou sémantikou môžu byť kombinované a rozšírené, a rozšírenia môžu vyžadovať voľnosť.

pomenovanie a zovšeobecnenie (naming and generics) – pomenovanie špecifikácie v jazyku CASL poskytuje dva hlavné ciele (odhliadnúc od čisto neformálneho cielu návrhu navrhovateľa špecifikácie): na vyhnutie sa doslovnému opakovaniu sa rovnakej časti špecifikácie vrámci jednej špecifikácie; a na povolenie jej vloženia do knižnice špecifikácií, aby sa tak špecifikácia mohla znovapoužiť jednoduchým odkazom jej mena v každej ďalšej neskoršej špecifikácii.

3.2 Syntax jazyka CASL

Najprv si pripomenieme, ako môžme formalizovať štruktúru špecifikácií v *inštitúciách* a *dokazovacom systéme*. Základná špecifikácia môže byť charakterizovaná nasledovne:

- trieda **Sig** *signatúr* Σ, každá determinuje symboly, ktorých určenie má byť špecifikované s *morfizmom* medzi signatúrami
- trieda **Mod**(Σ) *modelov*, s *homomorfizmom* medzi nimi, pre každú signatúru Σ
- množina **Sen**(Σ) *viet* (alebo *axiómov*) pre každú signatúru Σ
- relácia $\models$ *splnenia* medzi modelmi a vetami nad rovnakou signatúrou
- *dokazovací systém* na odvodenie viet z množiny viet

Morfizmus signatúry $\sigma : \Sigma \rightarrow \Sigma'$ určuje *translačnú funkciu* $\mathbf{Sen}(\sigma)$ na vetách, zobrazujúc $\mathbf{Sen}(\Sigma)$ na $\mathbf{Sen}(\Sigma')$, a *redukčnú* funkciu $\mathbf{Mod}(\sigma)$ na modely, zobrazujúc $\mathbf{Mod}(\Sigma')$ na $\mathbf{Mod}(\Sigma)$. Je potrebné dodržanie relácie splnenia pri preklade: $\forall\, S \in \mathbf{Sen}(\Sigma), M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \Leftrightarrow M' \models \mathbf{Sen}(\sigma)(S)$$

Od dokazovacieho systému sa požaduje aby mal význam (bol logický), tj., vety odvodené zo špecifikácie sú stále následné; okrem toho, dôsledok musí byť zachovaný pri preklade.

3.2.1 Základné konštrukcie

V tejto časti si popíšeme abstraktnú syntax a stanovenú interpretáciu konštrukcií vo viacdruhovej základnej špecifikácii.

```
BASIC-SPEC        ::=  basic-spec BASIC-ITEM*
```

Dobre vytvorená *(well-formed)* viacdruhová základná špecifikácia `BASIC-SPEC` jazyka CASL určuje základnú špecifikáciu základu viacdruhovej inštitúcie, pozostávajúc zo signatúry a množiny viet. Modely tejto signatúry a množiny viet poskytujú *sémantiku* základnej špecifikácie.

```
BASIC-ITEM        ::=  SIG-DECL | VAR-DECL | AXIOM
                    |  SORT-GEN
                    |  DATATYPE-DECL | ATTRIBUTION
```

Deklarácia signatúry `SIG-DECL` určuje časť signatúry. Deklarácia premennej `VAR-DECL` určuje zotriedené premenné, ktoré sú implicitne univerzálne kvantifikované v axiómach uzatvorenej `BASIC-SPEC` (premenné môžu byť taktiež deklarované lokálne explicitným kvantifikovaním v axiómach). Za zmienku stojí, že deklarácia premenných neprispieva k signatúram špecifikácie, v ktorej sa vyskytujú. `AXIOM` určuje vety tak, ako ohraničenie generátora druhov `SORT-GEN`. `DATATYPE-DECL` určuje časť signatúry (druh, spolu s istými konštruktormi a selektormi funkcií), tak ako niektoré vety (definujúce hodnoty selektorov).
`ATTRIBUTION` určuje len niektoré vety (vlastnosti výrokov ako asociatívnosť a komutatívnosť konkrétnych funkcií).

Deklarácia signatúr

```
SIG-DECL          ::=  SORT-DECL | FUN-DECL
                    |  PRED-DECL
```

Deklarácia druhu `SORT-DECL` určuje jeden alebo viac druhov. Deklarácie funkcie `FUN-DECL` určuje jednu alebo viac funkčných symbolov určitého profilu; podobne pre deklaráciu predikátov `PRED-DECL`. Funkčné a predikátové symboly môžu byť preťažené tak, že budú deklarované s viacerými profilmi v rovnakej

základnej špecifikácii. Deklarácia viacerých symbolov spolu v jednom `DECL` je ekvivalentná ich oddelenej deklarácii v skupine podobných `DECLs`.

```
ID              ::=  SIMPLE-ID
SIMPLE-ID       -- structure insignificant
```

Vnútorná štruktúra jednoduchého identifikátora `SIMPLE-ID`, určeného na identifikáciu druhov, funkcií a predikátov nie je dôležitá. Rovnaký identifikátor `ID` môže byť súčasne použitý na identifikovanie rôznych druhov entít, napr. druhov, funkcií a predikátov v rovnakej `BASIC-SPEC`.

Druhy

```
SORT-DECL        ::=  sort-decl SORT+
SORT             ::=  ID
```

Deklarácia druhu `SORT-DECL` určuje, že každý z druhov v skupine `SORT+` je v signatúre.

Konštanty a funkcie

```
  FUN-DECL         ::=  fun-decl FUN-NAME+ FUN-TYPE
! FUN-TYPE         ::=  SORT | PARTIAL-CON-TYPE |
!                    |  TOTAL-FUN-TYPE
!                    |  PARTIAL-FUN-TYPE
! PARTIAL-CON-TYPE ::=  partial-con-type SORT
! TOTAL-FUN-TYPE   ::=  total-fun-type SORT+ SORT
! PARTIAL-FUN-TYPE ::=  partial-fun-type SORT+ SORT
  FUN-NAME         ::=  ID
```

Deklarácia funkcie `FUN-DECL` určuje, že každé meno funkcie v zozname `FUN-NAME+` je v signatúre, v množine ciastočných alebo úplných funkčných symbolov a s argumentom druhu `SORT+` a výsledku `SORT` ako aj špecifikované funkčným typom

FUN-TYPE. FUN-TYPE pozostávajúci len z SORT je na deklaráciu úplných konštánt, ktoré zodpovedajú funkciám bez argumentov. PARTIAL-CON-TYPE slúži na deklarovanie čiastočných konštánt.

Predikáty

```
PRED-DECL        ::=  pred=decl PRED-NAME+ PRED-TYPE
  PRED-TYPE        ::=  pred-type SORT*
  PRED-NAME        ::=  ID
```

Deklarácia predikátov PRED-DECL určuje, že každé meno predikátu v zozname PRED-NAME+ je v signatúre, spolu so špecifikovaným argumentom druhu podľa typu predikátu PRED-TYPE.

Deklarácia premenných

```
  VAR-DECL         ::=  var-decl VAR+ SORT
  VAR              ::=  SIMPLE-ID
```

Deklarácia premenných VAR-DECL stanovuje zotriedené premenné pre použitie v axiómach; neprispieva k signatúre. Deklarácia premennej ako BASIC-ITEM skracuje pridávanie univerzálneho kvantifikátora premennej každej axióme v uzavretej BASIC-SPEC. Univerzálna kvantifikácia premennej, ktorá sa nevyskytuje voľne v axióme, je sémanticky irelevantná.

Axiómy a termy

```
  AXIOM            ::=  FORMULA
```

Axióma je uzavretá atomická formula viazaná kvantifikátorom. Axióma je dobre vytvorená keď je konštruovaná z dobre vytvorenej atomickej formuly (s ohľa-

dom na lokálne prostredie), a naviac všetky premenné, ktoré sa v nej vyskytnú sú deklarované v uzáverových kvantifikáciách alebo v deklarácii premennej uzavretej základnej špecifikácie. Dobre vytvorená axióma určuje vetu zo základnej špecifikácie nad danou množinou (underlying basic specification) (uzavretej univerzálnou kvantifikáciou nad deklarovanými premennými).

```
FORMULA          ::=  QUANTIFICATION | CONJUNCTION
                    |  DISJUNCTION | IMPLICATION
                    |  EQUIVALENCE | NEGATION | ATOM
```

Formula je vytvorená z atomickej formuly formy `ATOM` využívajúc kvantifikovanie a klasické logické spájanie.

Logické spojky

```
CONJUNCTION      ::=  conjunction FORMULA+
DISJUNCTION      ::=  disjunction FORMULA+
IMPLICATION      ::=  implication FORMULA FORMULA
EQUIVALENCE      ::=  equivalence FORMULA FORMULA
NEGATION         ::=  negation FORMULA
```

Tieto formuly určujú bežné logické spojky na subformuliach. Konjunkcia a disjunkcia sú zovšeobecnené do zoznamu formúl `FORMULA+`, a sú dobre vytvorené len keď zoznam obsahuje aspoň dve formuly.

Kvantifikátory

```
QUANTIFICATION   ::=  quantification QUANTIFIER
                      VAR-DECL+ FORMULA
QUANTIFIER       ::=  forall | exists
                    |  exists-uniquely
```

Kvantifikácia s viac ako jednou deklaráciou premennej `VAR-DECL` skracuje zväzok kvantifikácií s tým istým kvantifikátorom. Kvantifikácia s deklaráciou deklarujúc viac ako jednu premennú skracuje kvantifikáciu s tým istým kvantifikátorom a druhom. Platnosť deklarácie premennej v `QUANTIFICATION` je komponent `FORMULA`, a vnútorná deklarácia pre premenné s rovnakým identifikátorom ako vo vonkajšej deklarácii prekrýva vonkajšiu deklaráciu.

Atomické formuly a termy

```
  ATOM              ::=  TRUTH | PREDICATION
!                     |  DEFINEDNESS
!                     |  EXISTL-EQUATION
!                     |  STRONG-EQUATION
```

Atomická formula je dobre vytvorená (s ohľadom na lokálne prostredie a deklaráciu premenných) ak druh je dobre vytvorený (well–sorted) a rozširuje sa do jedinečnej atomickej formuly vytvárajúc vetu. Vzhľadom na preťaženie predikátov a/alebo funkčných symbolov, dobre vytvorený druh atomickej formule alebo termu môže mať viac rozšírení, zabraňujúc jeho dobrej vytvorenosti. Kvantifikácie funkcií a predikátových symbolov môžu byť použité na určenie mienených rozšírení a robí ich dobre vytvorenými; explicitné druhy argumentov a/alebo výsledkov môžu taktiež pomôcť vyhnúť sa nemienených rozšírení.

```
TRUTH               ::=  true | false
```

Atomická formula `true` a `false` sú stále dobre vytvoreným druhom a expandujú na primitívne vety, ako napr. vety pre `true` sú pravdivé a pre `false` sú nepravdivé.

```
PREDICATION         ::=  predication PRED-SYMB TERM*
PRED-SYMB           ::=  pred-symb PRED-NAME PRED-TYPE?
```

Použitie predikátového symbolu PRED-SYMB je dobre vytvorený druh, ak je deklarácia PRED-SYMB (so špecifickým PRED-TYPE, ak je) ako napr. všetky termy argumentov sú dobre vytvoreným druhom na deklarovanie druhov argumentov.

```
  DEFINEDNESS       ::=  definedness TERM
```

Definovateľná formula je dobre vytvorený druh, ak term je dobre vytvorený druh pre rovnaký druh.

```
! EXISTL-EQUATION   ::=  existl-equation TERM TERM
! STRONG-EQUATION   ::=  strong-equation TERM TERM
```

Porovnanie je dobre vytvorený druh ak existuje druh, že oba termy sú dobre vytvorené druhy pre tento druh. Potom expanduje na zodpovedajúcu existenčnú alebo silnú rovnicu na plne kvalifikovanom rozšírení termu pre tento druh.

```
! TERM              ::=  VAR | CONSTANT | APPLICATION
                      |  SORTED-TERM
```

Premenná VAR je dobre vytvorený druh pre druh špecifikovaný v jej deklarácii.

```
! CONSTANT          ::=  FUN-SYMB
! APPLICATION       ::=  application FUN-SYMB TERM+
  FUN-SYMB          ::=  fun-symb FUN-NAME FUN-TYPE?
```

CONSTANT je dobre vytvorený druh ak FUN-SYMB je deklarovaný ako konštanta (s príslušným FUN-TYPE, ak je špecifikovaný). APPLICATION funkčného symbolu FUN-SYMB je dobre vytvorený druh ak existuje deklarácia FUN-SYMB (s príslušným FUN-TYPE, ak je špecifikovaný) tak, že všetky termy argumentov sú dobre vytvorené druhy na deklarovanie druhov argumentov.

```
  SORTED-TERM       ::=  sorted-term TERM SORT
```

Term udaný so svojím druhom je dobre vytvorený druh, ak komponent term je dobre vytvorený druh pre špecifický druh.

Generovanie druhov

```
SORT-GEN          ::=  sort-gen SORT+ FUN-SYMB+
```

Obmedzenia generátora druhov `SORT-GEN` sú dobre vytvorené, keď všetky druhy a funkčné symboly špecifikované v ňom sú deklarované v lokálnom prostredí (sám nedeklaruje žiaden druh ani funčný symbol). Potom určuje príslušnú vetu. Tu `FUN-SYMB`, ktorý je len `FUN-NAME` (bez explicitného `FUN-TYPE`) skracuje zoznam všetkých deklarovaných funkčných symbolov, ktorých meno má rovnaký identifikátor ako špecifikovaný `FUN-NAME`.

Deklarácia dátových typov

```
! DATATYPE-DECL    ::=  datatype-decl SORT ALTERNATIVE+
! ALTERNATIVE      ::=  CONSTANT | CONSTRUCT
```

Deklarácia dátového typu `DATATYPE-DECL` určuje jej komponent `SORT`. Pre každú alternatívu `CONSTRUCT` deklaruje požadované funkcie – konštruktor a selektor, a určuje axiómy výrokov očakávaného vzťahu medzi nejakým selektorom a konštruktorom. Všetky druhy použité v každej `ALTERNATIVE` musia byť deklarované v lokálnom prostredí poskytujúc uzavretú `DATATYPE-DECL`.

```
! CONSTRUCT        ::=  construct FUN-NAME COMPONENT+
```

Funkcia `FUN-NAME` špecifikovaná v `CONSTRUCT` je deklarovaná ako úplný konštruktor. Každý `COMPONENT` špecifikuje jeden argument druhu pre konštruktor; jej výsledným druhom je `SORT` uzavretej `DATATYPE-DECL`.

Keď všetky ALTERNATIVE konštrukcie sú CONSTANTs, definovaný typ zodpovedá (neusporiadanému) vymenovanému (enumeračnému) typu.

```
! COMPONENT        ::=  component FUN-NAME? SORT
```

COMPONENT bez FUN-NAME poskytuje iba svoj SORT ako argument druhu pre konštruktor pre uzavretý CONSTRUCT.
COMPONENT s FUN-NAME navyše deklaruje funkciu FUN-NAME ako selektor s argumentom druhu daný uzavretou DATATYPE-DECL a s naznačovaným výsledným druhom. Funkcia je deklarovaná ako úplná ak existuje len jedna ALTERNATIVE v uzavretej DATATYPE-DECL ináč ako čiastočná.

Atribúty

```
! ATTRIBUTION      ::=  attribution PROPERTY+ FUN-SYMB+
! PROPERTY         ::=  SIMPLE-PROPERTY | UNIT-PROPERTY
! SIMPLE-PROPERTY  ::=  associative | commutative
                     |  idempotent
! UNIT-PROPERTY    ::=  unit-property FUN-SYMB
```

ATTRIBUTION nejakej vlastnosti funkčnému symbolu FUN-SYMB+ určuje jasné vetné tvrdenia, že každá z funkcie bude mať každú zo špecifikovaných vlastností. Tu FUN-SYMB, ktorý je len FUN-NAME (bez explicitného FUN-TYPE) skracuje zoznam všetkých funkčných symbolov deklarovaných v lokálnom prostredí, ktorých meno má rovnaký identifikátor ako špecifikované FUN-NAME. ATTRIBUTION je dobre vytvorený práve vtedy, ak axiómy ktoré skracuje sú dobre vytvorené – toto vo všeobecnosti vyžaduje, aby každý špecifikovaný funkčný symbol bol deklarovaný (niekde inde) aby preberal dva argumenty, obe rovnakého druhu, a aby symbol špecifikovaný v UNIT-PROPERTY bol deklarovaný (niekde inde) ako konštanta príslušného druhu.

3.3 Haskell – funkcionálny jazyk

Haskell je funkcionálny programovací jazyk. V detaile je to polymorfne typovaný, lenivý (lazy), čisto funkcionálny jazyk (čisto funkcionálny sa myslí na časť, ktorou sa budeme zaoberať). Haskell je založený na lambda calculuse[4]. Haskell taktiež poskytuje funkcie vyššieho rádu, nestriktnú sémantiku, statické polymorfné typovanie, používateľom definované algebraické dátové typy, pattern–matching, systém modulov, jednočlenný systém vstupu/výstupu, bohatú množinu primitívnych dátových typov, vrátane zoznamov, polí, čísla s ľubovoľnou a pevnou presnosťou a čísla s pohyblivou rádovou čiarkou[3].

Mnoho času v životnom cykle softvérových produktov je trávený na špecifikovanie, návrh a údržbu, nie na programovanie. Funkcionálne jazyky sú skvelé na písanie špecifikácií, ktoré môžu byť dokonca vykonávané (a od tejto chvíle aj testované a ladené). Takáto špecifikácia sa potom stáva prvým prototypom finálneho programu.

3.3.1 Typy

Pretože Haskell je čisto funkcionálny jazyk, všetky výpočty sú vykonávané cez vyhodnocovanie výrazov (syntaktických termov) na oznámenie hodnoty (abstraktnej entity, ktorú považujeme za výsledok). Každá hodnota má asociovaný typ. Tak ako výrazy označujú hodnoty, typové výrazy sú syntaktické termy, ktoré určujú typ hodnoty (alebo len typy). Funkcie v Haskell–i sú normálne definované sériou rovníc. Rovnica je príkladom deklarácie. Inou možnosťou deklarácie je deklarácia typovej signatúry, pomocou ktorej môžme explicitne deklarovať typ funkcie. Statický typový systém Haskellu[1] definuje formálny vzťah medzi typmi a hodnotami. To zabezpečuje, že programy písane v Haskell–i, sú typovo bezpečné. Hlavnou výhodou staticky typových jazykov je, že všetky typové chyby sú detekované v čase kompilácie.

Haskell taktiež začleňuje polymorfické typy — typy, ktoré sú univerzálne

kvantifikované v určitom smere nad všetkými typmi. Polymorfický typový výraz v podstate popisuje rodiny typov. Identifikátory, ktoré takéto rodiny typov označujú, nazývame typovými premennými a implicitne sú univerzálne kvantifikované. Príkladom pre vysvetlenie princípov polymorfizmu je dátová štruktúra "zoznam". S ohľadom na tento prejav generalizácie, typový systém Haskell–u vykazuje dve dôležité vlastnosti:

- každý dobre napísaný výraz má zabezpečený jedinečný principiálny typ
- principiálny typ môže byť automaticky odvodený

V Haskell–i môžme taktiež definovať naše vlastné typy použitím deklarácie `data`. Často sa využíva pri definovaní nových typov *typový konštruktor* a na definovanie možných hodnôt typu *dátový konštruktor*. Je dôležité, aby sme rozlišovali medzi aplikovaním *dátového konštruktora* na oznámenie hodnoty, a aplikovaním *typového konštruktora* na oznámenie typu. Prvá časť sa deje za behu a znamená ako počítame veci v Haskell–i, zatiaľ čo druhá časť sa deje v čase kompilácie a je časťou procesu typového systému na zabezpečenie ochrany typu.

3.3.2 Funkcie

Funkcie zohrávajú vo funkcionálnom jazyku hlavnú úlohu. Namiesto použitia rovníc na definovanie funkcií, môžme ich taktiež definovať "anonymne" pomocou *lambda abstrakcie*. Funkcie su nestriktné. Abstraktne vyjadrené, označujeme hodnotu neterminujúceho výrazu ako $\perp$ (bottom). Výrazy, ktoré nadobúdajú výsledok v určitom druhu chyby, ako napr. $1/0$, majú tiež túto hodnotu. Takéto chyby nie sú zotaviteľné: program nebude pokračovať vo vykonávaní za týmyto chybami. Nestriktné funkcie sa taktiež nazývajú "lenivé funkcie" (lazy functions) a vyhodnocujú svoje argumenty "lenivo" alebo "podľa potreby". Hlavnou výhodou toho je, že výpočtovo náročné hodnoty môžu byť predané ako argument funkcii bez obavy, že budú vypočítané, ak nie sú potrebné. Dôležitým príkladom tohto je možná nekonečná dátová štruktúra.

3.3.3 Typové triedy a preťaženie

Funkcionálny jazyk Haskell obsahuje dva druhy polymorfizmu: *parametrický* a *ad hoc*. Ad hoc polymorfizmus je tiež známy ako preťaženie. V Haskell–i typové triedy poskytujú štrukturovaný spôsob na kontrolovanie ad hoc polymorfizmu alebo preťaženia. Typové triedy obyčajne riešia oba problémy:

- dovoľujú nám deklarovať aké typy sú inštancie ktorej triedy
- poskytujú definície o preťažených operátoroch asociovaných s triedou

Haskell taktiež podporuje *rozšírenia* tried a mnohonásobné dedenie. V týchto triedach sa používajú ako argumenty typy prvého rádu, ale taktiež sa dajú použiť aj typové konštruktory, ktoré berú typ ako argument a vracajú typ ako výsledok. V Haskell–i neexistujú také hodnoty, ktoré majú takýto typ, ale tieto typy vyššieho rádu môžu byť použité pri deklarovaní tried. Toto deklarovanie inštancie je veľmi užitočnou schopnosťou a demonštruje možnosť popisu všeobecného "kontajnerového" typu, povoľujúc funkciám (napr. `map`) aby pracovali uniformne nad ľubovonými dátovými typmi (stromy, zoznamy, atď.).

Haskell má ešte jeden typový systém, ktorým zabezpečuje korektnosť typov a zisťuje chybné typové výrazy. Každý typ ma asociovanú kategóriu, ktorá zabezpečuje, že typ je použitý korektne. Typové výrazy sú klasifikované do rôznych kategórií, ktoré nadobúdajú dve možné formy:

- symbol $\star$ reprezentuje kategóriu asociovaného typu s konkrétnym dátovým objektom
- ak κ_1 a κ_2 sú kategórie, potom $\kappa_1 \rightarrow \kappa_2$ sú kategóriou typov, ktoré nadobudnú kategóriu typu κ_1 a vrátia kategóriu typu κ_2

Kategórie sa neobjavujú priamo v programoch, ale ostavajú v pozadí, okrem prípadu, ak chybová typová signatúra vedie ku kategorizačnej chybe.

4 Zobrazenie

V súčasnosti existuje mnoho algebraických špecifikácií. Tabuľka 4 – 1 zobrazuje hlavné systémy (framework) s hrubou chronológiou. Niektoré z nich sú grandiózne, širokospektrálne systémy, vybavené s úplnou vývojovou metodológiou softvéru. Iné sú viac prosté, obsahujúc zásadne prototypovacie a verifikačné nástroje a ich asociované jazyky.

Abel Etl Spectral Spectrum Lpg	
Extended-ML Obscure Troll	1990
Act-Two Asf+Sdf Rsl	
Asl Larch Rap SMoLCS	
Asspegique Cold-K	1980
Act-One Pluss	
Cip Obj	
Clear	1970
Look	

Tabuľka 4 – 1 systémy algebraických špecifikácií

Väčšina algebraických systémov bola vyvinutá na fakultách príslušných univerzít alebo medzinárodnou spoluprácou individuálnych výskumníkov a každý zo systémov slúži skôr lokálne. Nedostatok spoločného, široko podporovaného systému pre algebraické špecifikácie je hlavným problémom. Preto vznikla aj iniciatíva skupiny CoFI k návrhu spoločného jazyka. Jazyk CASL preberá z množstva jazykov viacere konštrukcie pre spätnú kompatibilitu.

Jazyk CASL sme vybrali pre jeho viaceré vlastnosti. Jeho hlavné črty:

- návrhovo založený na kritickom výbere koncepcií a konštrukcií nachádzajúcich sa v existujúcich systémoch

- jednoduchá sémantika a dobrá pragmatika[3]
- hlavná časť rodiny súvisiacich jazykov, ktoré môžu byť použité ako podjazyky, alebo rozšírenia

CASL taktiež pozostáva s viacerých hlavných častí, ktoré sú na sebe nezávislé a môžu byť pochopené (a použité) oddelene.

4.1 Konštrukcia špecifikácie požiadaviek

V špecifikácii požiadaviek si najprv musíme položiť otázku, *čo* chceme urobiť. Najprv formulujeme otázku v prirodzenom jazyku. Pretože pravdivosť riešenia musíme dokázať použitím logiky, musíme nájasť predpoklady otázky ako pravdivé vety a formulovať ich v nejakom formálnom jazyku s matematicky definovanou sémantikou. Ilustrujeme to na príklade 4.1.

Príklad 4.1 *špecifikácia požiadaviek zoznamu:*

```
spec Elem=
  sort Elem

spec List[Elem]=
  free type
    List[Elem] ::= nil | cons(first:?Elem; rest:?List[Elem])
  op
     __++__ : List[Elem]×List[Elem]→List[Elem], assoc, unit nil
     head : List→?Elem;
     tail : List→?List;
  vars
     e:Elem; l,l':List[Elem]
  axiom
     cons(e,l)++l' = cons(e,l++l')
     not(def(head(nil)))
```

[3]berúci do ohľadu praktické využitie

```
not(def(tail(nil)))
head(cons(e,l))=e
tail(cons(e,l))=l
```

Špecifikácia popisuje *abstraktný údajový typ*. Schéma údajovej abstrakcie, ktorú nazývame *signatúrou*, je konštrukciou univerzálnej algebry. Predpokladáme, že údajové abstrakcie a tým aj signatúry sú realizované ako jazykové konštrukcie. Každú špecifikáciu môžme popísať odpovedajúcou algebraickou špecifikáciou ako:

$$Spec = (\Sigma, \Psi_\Sigma)$$

kde Σ je signatúra (viď. kapitola 2.1) a Ψ_Σ je množina Σ–axiómov, tj. uzavreté Σ–formuly (Σ–vety) napísané v jazyku predikátovej logiky prvého rádu. Σ–axiómy sú axiómy použité v matematických teóriách, tj. dokázané teorémy axiomatickej teórie množín. Popisujú predpoklady špecifikácie požiadaviek, tj. nejaké vlastnosti funkcií zodpovedajúce funkčným symbolom zo signatúry Σ.

Každej signatúre Σ vieme zostrojiť Σ–algebru pridelením množiny dát, funkcií a pravdivých Σ–viet druhom, resp. funkčným symbolom a predikátovým symbolom. Ak pre Σ–vetu φ existuje dôkaz v Σ–algebre M, hovoríme, že φ *je splnené v* M, zapísané ako

$$M \models_\Sigma \varphi$$

a takáto Σ–algebra M so splnenými Σ–vetami nazývame Σ–modelom. Každej signatúre Σ zo **Sign** vytvoríme kategóriu **Mod(**Σ**)** pozostávajúcu zo Σ–modelov ako objekty kategórie a redukovaný homomorfizmus medzi nimi ako morfizmus kategórií.

Ďalej potrebujeme dva funktory

$$Sen : \mathbf{Sign} \to \mathbf{Set} \quad a \quad Mod : \mathbf{Sign^{op}} \to \mathbf{Cat}$$

Funktor *Sen* priraďuje každej signatúre Σ z **Sign** množinu pravdivých Σ–viet z kategórie množín **Set**. Funktor *Mod* z duálnej kategórie signatúr $\mathbf{Sign}^{\mathbf{op}}$ do kategórie **Cat** malých kategórií priraďuje každej signatúre Σ kategóriu Σ–modelov **Mod(**Σ**)**. Štvoricu

$$I = (\mathbf{Sign}, Sen, Mod, (\models_\Sigma)_{\Sigma \in \mathbf{Sign}})$$

nazývame *inštitúciou špecifikácie požiadaviek*[2]. Z vyššie uvedenej konštrukcie je jasné, že Σ–veta φ je splnená v Σ_1–modeli M_1 ak jej obraz pod funktorom *Sen* je Σ_2–veta splnená v redukovanom Σ_2–modeli M_2.

4.2 Konštrukcia špecifikácie návrhu

4.2.1 Signatúra

Každá signatúra špecifikácie návrhu pozostáva z

$$\Sigma' = (TO, \varepsilon, \leq, v^+, v^-, C)$$

kde *TO* je množina typových operátorov. Typové operátory môžu byť typovými konštruktormi alebo typovými synonymami (aliasmi). Typové hodnoty vo funkcionálnom jazyku sú vytvorené z typových konštruktorov. Typové výrazy sú klasifikované do rôznych kategórií. Všetky nulové typové konštruktory majú kategóriu označovanú symbolom "*". Druh je nulovým typovým konštruktorom. Ak κ_1 a κ_2 sú kategóriami, potom $\kappa_1 \rightarrow \kappa_2$ je kategóriou typov, ktoré zoberú typ kategórie κ_1 a vrátia typ kategórie κ_2. Uvedieme tri vstavané typové konštruktory:

súčin	$_ * _ : \kappa \rightarrow \kappa \rightarrow \kappa$
čiastočné funkcie	$_ \rightarrow ? _ : \kappa \rightarrow \kappa \rightarrow \kappa$
typ singleton	$unit : \kappa$

kde κ je kategória všetkých typov. Funkcionálne jazyky dovoľujú typové synonymá alebo aliasy, tj. mená pre všeobecne používané typy. Typové synonymá

nedefinujú nový typ, ale priraďujú nové mená už existujúcim typom. Každému typovému operátoru *F* z *TO* je priradená kategória $\kappa \rightarrow \kappa \rightarrow \ldots \rightarrow \kappa$.

Alias–free mapovanie ε priraďuje každému typovému synonymu *pseudotyp*. Pseudotyp je λ–term

$$\lambda a_1, a_2, \ldots a_n : \kappa \bullet t$$

kde $a_1, a_2, \ldots a_n$ sú typovými premennými a typ *t* obsahuje najviac $a_1, a_2, \ldots a_n$ typových premenných. Pretože rekurzívny rozvoj pre typové synonymá je zakázaný, každý typ *t* má jedinečné alias–free rozšírenie označené ako $\varepsilon(t)$.

Typy asociované signatúre Σ' môžu byť typovými premennými alebo typovými operátormi $F(t_1, t_2, \ldots t_n)$.

Povoľujeme typový polymorfizmus s podtypovými obmedzeniami, teda typové schémy sú pridané typom. Typová schéma je

$$\forall a_1, a_2, \ldots a_n : \kappa \bullet s_1 \sqsubseteq t_1, \ldots s_m \sqsubseteq t_m \Rightarrow t$$

Obmedzenia subtypovania $s_i \sqsubseteq t_i$, pre $i = 1, 2, \ldots m$ indikujú, že interpretácia typových premenných je obmedzená na také typy, pre ktoré táto relácia subtypu je platná. Je triviálne dokázať, že relácia subtypovania $' \sqsubseteq'$ je typu preorder, tj. je reflexívna a tranzitívna.

Každému typovému konštruktoru *F* pridáme predikát variability v^+ a v^-. Indikujú, ktoré argumenty typového konštruktora *F* môžu byť použité v mechanizme subtypu ako *covariant*(v^+) alebo *contravariant*(v^-). Keďže aj funkcie môžu byť predané ako argumetny iným funkciám, musíme dať nasledovné pravidlo subtypovania pre (funkčné) typy, tj. musíme špecifikovať za akých okolností je bezpečné použitie funkcie ako jedného typu v kontexte, kde je očakávaný iný typ:

$$\frac{t_1 \sqsubseteq s_1 \quad s_2 \sqsubseteq t_2}{s_1 \rightarrow s_2 \sqsubseteq t_1 \rightarrow t_2}$$

Ak t_1 je subtypom s_1, $t_1 \sqsubseteq s_1$, a s_2 je subtypom t_2, $s_2 \sqsubseteq t_2$, potom typ $s_1 \rightarrow s_2$ je subtypom typu $t_1 \rightarrow t_2$. Zmysel relácie subtypu je opačný (contravariant) pre argumenty typu v pravidle ľavého predpokladu, pokiaľ sa spracováva v rovnakom

smere (covariant) pre typy výsledkov ako aj pre funkčné typy. Ak máme funkciu *f* typu $s_1 \rightarrow s_2$, potom vieme, že *f* akceptuje elementy typu s_1; *f* tiež akceptuje elementy typu t_1. Môžme taktiež vidieť, že výsledok patrí hocakému supertypu t_2 z s_2. Hocaká funkcia *f* typu $s_1 \rightarrow s_2$ môže mať typ $t_1 \rightarrow t_2$. Alternatívny pohľad je, že je bezpečné povoliť funkcii typu $s_1 \rightarrow s_2$ aby bola použitá v kontexte, kde iný typ $t_1 \rightarrow t_2$ je očakávaný dovtedy, pokiaľ žiadny z argumentov, ktorý môže byť predaný funkcii v tomto kontexte, nebude prekvapujúci ($t_1 \sqsubseteq s_1$), a žiadny z výsledkov, ktoré vráti nebudú neočakávané kontextom ($s_2 \sqsubseteq t_2$). Napríklad, predikáty variabilnosti pre vstavané typové konštruktory pre súčin a parciálne funkcie sú:

$$\begin{array}{rclrcl} v^+(_ * _) & = & \{1,2\} & v^-(_ * _) & = & \emptyset \\ v^+(_ \rightarrow _) & = & \{2\} & v^-(_ \rightarrow _) & = & \{1\} \end{array}$$

kde čísla (elementy množiny) sú sériové čísla argumentov. Nech $s_1, s_2, \ldots s_n$, $t_1, t_2, \ldots t_n$ sú (alias–free) typmi a *F* nech je n–árny typový konštruktor. Potom relácia

$$F(s_1, s_2, \ldots s_n) \sqsubseteq F(t_1, t_2, \ldots t_n)$$

je platná, ak pre všetky $i = 1, \ldots n$

$$\begin{array}{rll} & i \in v^+(F) & \text{a} \quad s_i \sqsubseteq t_i \\ \text{alebo} & i \in v^-(F) & \text{a} \quad t_i \sqsubseteq s_i \end{array}$$

Množina *C* je množinou konštantných symbolov s priradenými typovými schémami. Konštantný symbol s menom *c* a profilom *t* je písaný $c : t$. *C* obsahuje taktiež rozlíšiteľnú konštantu

$$() : unit$$

potrebnú na mapovanie predikátových symbolov zo signatúry špecifikácie požiadaviek Σ.

Teraz môžme definovať morfizmus medzi signatúrami špecifikácie návrhu. Morfizmom signatúr

$$\sigma' : \Sigma'_1 \to \Sigma'_2$$

definujeme jednoducho morfizmom medzi ich komponentami:

- typové konštruktory zo Σ'_1 sú mapované na typové operátory v Σ'_2 zachovávajúc profil a predikáty variability; typové synonymá nie sú vôbec mapované
- konštantné symboly z Σ'_1 sú mapované na konštantné symboly v Σ'_2 rešpektujúc profil a subsorting.

Označíme znakom **Sign**′ štruktúru, pozostávajúcu z

- triedy signatúr špecifikácie návrhu
- množiny morfizmov medzi nimi.

Je triviálne dokázať, že kompozícia týchto morfizmov je asociatívna, tj. **Sign**′ je kategóriou a nazývame ju kategóriou signatúr špecifikácie návrhu.

4.2.2 Formula

Na vytvorenie druhého komponentu inštitúcie špecifikácie návrhu, tj. funktor

$$Sen' : \mathbf{Sign}' \to \mathbf{Set}$$

ktorý poskytuje každej signatúre špecifikácie návrhu Σ' množinu Σ–viet, musíme definovať pravidlá na vytváranie viet špecifikácie návrhu. Nepatrne sa líšia od viet požiadaviek. Definujeme pravidlo na vytvorenie termov, ktoré rozširujú možnosti zápisu termov o napr. λ–abstrakciu:

- premenná $x : t$ typu t je term

- ak $\alpha : s$ je term typu *s*, ktorý je subtypom typu *t*, $s \sqsubseteq t$, potom $\alpha : t$ je taktiež term typu *t*
- ak $\alpha_1, \alpha_2, \ldots \alpha_n$ sú termy typu $t_1, t_2, \ldots t_n$, potom $(\alpha_1, \alpha_2, \ldots \alpha_n) : t_1 * t_2 * \ldots * t_n$ je termom súčinu typu $t_1 * t_2 * \ldots * t_n$
- ak $\alpha : s \rightarrow ?t$ je termom typu čiastočnej funkcie a $\beta : s$ je term typu *s*, potom taktiež použitie $\alpha(\beta) : t$ je term typu *t*
- ak $\alpha : t$ je term a $x_1 : s_1, x_2 : s_2, \ldots x_n : s_n$ sú premenné korešpondujúceho typu, taktiež

  $$\lambda X_1 : s_1, x_2 : s_2, \ldots x_n : s_n \bullet \alpha : s_1 * s_2 * \ldots * s_n \rightarrow ?t$$

  je term

Vety dovoľujú, aby univerzálny a existenčný kvantifikátor mali rozsah aj nad typovými premennými $a : \kappa$ a mohli obsahovať obmedzenia subtypovanie v rovnakej forme ako typové schémy. Kvantifikácia nad typovými schémami je zakázaná. Predikáty vo vetách sú nahradené termami typu *unit*. Hoci neexistuje priamy postup na použitie logických spojok v λ–termoch, môžme písať napríklad konštantu pravdivostného predikátu typu *a* len ako $\lambda x : a \bullet ()$; a konjunkciu ako $\lambda x_1, x_2 : unit \bullet ()$.

4.2.3 Model

Tretím komponentom inštitúcie špecifikácie návrhu je funktor

$$Mod' : \mathbf{Sign}' \rightarrow \mathbf{Cat}$$

ktorý priraďuje každej signatúre návrhu Σ' kategóriu $\mathbf{Mod}(\Sigma')$ Σ'–modelov. Najviac vhodné sa zdá byť vybratie za model špecifikácie návrhu M' silný Henkinov model, kde funkcie sú interpretované ľubovoľnými množinami vybavených

s aplikačnými operáciami príslušného typu. Takéto modely dovoľujú lepšiu analýzu funkčných miest a priamo zodpovedajú s triedou modelov špecifikácie požiadaviek.

Nech Σ' je signatúra špecifikácie návrhu. Model špecifikácie návrhu M' je algebra, pozostávajúca z množiny M'_t pre každý alias–free typ *t* generovaný Σ' v takom smere, že

- *unit* je interpretovaný ako množina singleton
- typy súčinu $t_1 * t_2 * \ldots * t_n$ sú interpretované ako kartézsky súčin spolu s priradením čiastočnej interpretovacej funkcie

  $$M'_{t1} \times M'_{t2} \times \ldots \times M'_{tn} \rightarrow ? M'_t$$

  každému termu typu *t* s premennými $x_1 : t_1, x_2 : t_2, \ldots x_n : t_n$.

Tieto interpretovacie funkcie musia rešpektovať odvoditeľnú rovnosť termov, zvyčajné pravidlá rovnosti pre existenciálnu rovnosť a pre β– η– a ξ–pravidlá s viacerými argumentami v prípade paralelnej λ–abstrakcie nad viacerými premennými. Substitúcia musí byť modelovaná ako kompozícia čiastočných funkcií. Termy majú formu

$$\lambda x : s \bullet x : t$$

kde $s \sqsubseteq t$ sú interne úplné funkcie.

Morfizmi medzi Σ'–modelmi sú vačšinou homomorfizmami, tj. množina homomorfizmov pre každý typ *t*. Pre každú signatúru špecifikácie návrhu Σ' trieda Σ'–modelov spolu so zodpovedajúcim homomorfizmom medzi nimi vytvára kategóriu $\mathbf{Mod}(\Sigma')$ modelov špecifikácie návrhu.

Relácia splnitelnosti $\models_{\Sigma'}$ je v podstate rovnaká ako pre inštitúciu špecifikácie požiadaviek. Formuly, ktoré sú univerzálne kvantifikované nad typovými premennými s obmedzeniami subtypu sú splniteľné ak uzatvorená formula je

splniteľná pre všetky ohodnotenia typových premenných, ktoré splňujú obmedzenia. Ak výsledok aplikácie predikátu je definovaný, potom je splniteľný ako atomická formula.

Hovoríme, ze štvorica

$$\mathbf{I}' = (\mathbf{Sign}', Sen', Mod', (\models_{\Sigma'})_{\Sigma' \in \mathbf{Sign}'})$$

je inštitúcia špecifikácie návrhu.

4.3 Prechod od špecifikácie požiadaviek k špecifikácii návrhu

Teraz vytvoríme prvý krok procesu vývoja programu v rámci inštitúcií, tj. inštitučnú šipku od inštitúcie špecifikácie požiadaviek **I** k inštitúcii špecifikácie návrhu $\mathbf{I}'$. Táto šipka pozostáva z nasledujúcich mapovaní. Prvé je funktor ϕ, ktorý mapuje kategóriu špecifikácie požiadaviek **Sign** na kategóriu špecifikácie návrhu $\mathbf{Sign}'$

$$\phi : \mathbf{Sign} \rightarrow \mathbf{Sign}'$$

tak, že pre každú signatúru špecifikácie požiadaviek $\Sigma = (S, O, P)$, funktor ϕ mapuje jej komponenty nasledovne:

- druhy z *S* sú mapované na nulový typový konštruktor *TO*
- konštantné funkčné symboly sú mapované na konštanty v *C*, ktoré obsahujú taktiež špeciálnu konštantu $() : unit$
- funkčné symboly z *O* sú mapované na typové operátory v *TO*. Ak profil funkčného symbolu obsahuje subtyp, musíme definovať predikáty premenlivosti pre neho. Menu typu pridáme zodpovedajúci λ–term, ktorý ho definuje
- predikátové symboly z *P* sú mapované na typové operátory v *TO*, s výsledným typom *unit*.

Druhým komponentom inštitučnej šipky je naturálna transformácia (tj. mapovanie medzi funktormi)

$$\mu^{Sen} : Sen' \circ \phi \rightarrow Sen$$

ktorá je množinou morfizmov pre každú signatúru Σ zo **Sign**. Σ–axiómy sú mapované na zodpovedajúce (prevažne) λ–termy spomenuté vyššie.

Posledným komponentom inštitučnej šipky je naturálna transformácia

$$\mu^{Mod} : Mod \rightarrow Mod' \circ \phi$$

prideľujúca modelu špecifikácie požiadaviek zodpovedajúci model špecifikácie návrhu. Potom inštitučná šipka je trojica

$$(\phi, \mu^{Sen}, \mu^{Mod}) : \mathbf{I} \rightarrow \mathbf{I}'$$

Na ilustrovanie tohto kroku je uvedený príklad 4.2.

Príklad 4.2 *špcifikácia návrhu zoznamu:*

```
cons          :: a → [a] → [a]
cons y []     = cons [y]
cons y (x:xs) = x:(cons y xs)

(++)         :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

head       :: [a] → a
head (x:_) = x
head []    = ⊥

tail        :: [a] → [a]
tail (_:xs) = xs
tail []     = ⊥
```

5 Príklady

V tejto podkapitole si uvedieme príklad číselného zásobníka, všeobecného zásobníka a stromu. Za základ celočíselného zásobníka slúžia celé čísla. Taktiež sme vybrali niekoľko základných operácií nad zásobníkom, konkrétne:

push – vloží hodnotu na vrch zásobníka

pop – zahodí hodnotu z vrchu zásobníka

top – vráti hodnotu z vrchu zásobníka

isEmpty – zistí, či je zásobník prázdny

Príklad 5.1 *špecifikácia požiadaviek číselného zásobníka:*

```
spec STACK[NUM]=
  free type
    Stack[Num] ::= nil
  op
     push      : Num→Stack;
     pop       : Stack→?Stack;
     top       : Stack→Num;
     isEmpty   : Stack→Bool;
  vars
     e:Num;  s:Stack[Num]
  axiom
     not(def(pop(nil)))
     not(def(top(nil)))
     pop(push(e,s))=s
     isEmty(nil)=true
```

K tejto špecifikácii uvedieme aj implementáciu vo funkcionálnom jazyku Haskell, pričom v tomto prípade je `Stack` len synonymom k zoznamu čísel.

Príklad 5.2 *číselný zásobník v jazyku Haskell:*

```
type Stack      = [Int]

push            :: Int → Stack → Stack
push x stk      = x:stk

top             :: Stack → Int
top (x:stk)     = x

pop             :: Stack → Stack
pop (x:stk)     = stk

isEmpty         :: Stack → Bool
isEmpty []      = True
isEmpty (x:xs)  = False
```

V prípade všeobecného zásobníka požadujeme, aby sme do neho mohli vkladať hocaké hodnoty.

Príklad 5.3 *špecifikácia požiadaviek všeobecného zásobníka:*

```
spec STACK[ELEM]=
  free type
    Stack[Elem] ::= nil
  op
     push      : Elem→Stack;
     pop       : Stack→?Stack;
     top       : Stack→Elem;
     isEmpty   : Stack→Bool;
  vars
     e:Elem; s:Stack[Elem]
  axiom
     not(def(pop(nil)))
     not(def(top(nil)))
     pop(push(e,s))=s
     isEmty(nil)=true
```

Keďže sa jedná o všeobecný zásobník, môžme do neho vkladať buď čísla alebo znaky, ale v zásade nemôžme miešať jednotlivé typy, keďže pri inicializácii zásobníka v podstate určíme, aký typ zásobníka to bude.

Príklad 5.4 *všeobecný zásobník v jazyku Haskell:*

```
data Stack a    = [a]

push            :: a → Stack a → Stack a
push x stk      = x:stk

top             :: Stack a → a
top (x:stk)     = x

pop             :: Stack a → Stack
pop (x:stk)     = stk

isEmpty         :: Stack a → Bool
isEmpty []      = True
isEmpty (x:xs)  = False
```

V poslednom príklade uvedieme binárny vyhľadávací strom. Stromy sú pri programovaní využívané často na rôzne účely, hlavne na rôzne druhy uchovávania dát a následne ich vyhľadávanie. Binárny strom je vo všeobecnosti špecifikovaný ako uzol s dvoma podstromami. Tento variant stromu je veľmi všeobecný, keďže pri binárnych stromoch sa väčsinou používa vyvažovanie stromu, na zachovanie čo najlepšej konzistentnosti.

Príklad 5.5 *špecifikácia požiadaviek binárneho vyhľadávacieho stromu:*

```
spec BSTREE[ELEM]=
  free type
    bsTree[Elem] ::= nil | Bin(Elem, bsTree[Elem], bsTree[Elem])
  op
     size      : bsTree→Num;
```

```
    height   : bsTree→Num;
    empty    : bsTree;
    insert   : Elem×bsTree→bsTree;
    member   : Elem×bsTree→Bool;
  vars
    e:Elem; s:bsTree[Elem]
  axiom
    size(nil)=0
    height(nil)=0
    empty=nil
    insert(e,nil)=Bin(e,nil,nil)
    member(e,nil)=false
```

V našom binárnom vyhľadávacom strome sme implementovali štandardné funkcie:

size – zistí veľkosť stromu

height – zistí výšku stromu[4]

empty – vytvorenie prázdneho stromu

insert – vloženie prvku do stromu

member – zistí, či daný prvok sa nachádza v strome[5]

Príklad 5.6 *binárny vyhľadávací strom:*

```
Bin            :: a → (bsTree a) → (bsTree a) → (bsTree a)
data bsTree a = Nil | Bin a (bsTree a) (bsTree a)

size                :: (bsTree a) → Num
size Nil            = 0
size (Bin x lt rt) = 1 + sizebst lt + sizebst rt
```

[4]**height** sa počíta len s väčším podstromom, pričom **size** zaratáva oba podstromy
[5]strom sa prakticky rekurzívne prehľadáva podľa daných podmienok

```
height                :: (bsTree a) → Num
height Nil            = 0
height (Bin x lt rt) = 1 + (height lt ‘max‘ height rt)

empty  :: (bsTree a)
empty = Nil

insert                  :: a → (bsTree a) → (bsTree a)
insert x Nil            = Bin x Nil Nil
insert x (Bin y t1 t2) = Bin y (insert x t1) t2, if x < y
                        = Bin y t1 t2           , if x = y
                        = Bin y t1 (insert x t2), if x > y

member                  :: a → (bsTree a) → bool
member x Nil            = False
member x (Bin y t1 t2) = member x t1, if x < y
                        = True       , if x = y
                        = member x t2, if x > y
```

Medzi štandardné funkcie patrí taktiež aj funkcia **delete**, ale vzhľadom na jej zložitosť nie je uvedená.

6 Zhodnotenie riešenia

Proces vývoja jazykov viedol od riešenia numerických problémov po symbolické spracovanie matematických formúl. V súčasnosti sa vývoj uberá smerom k návrhu špecifikačných a programovacích jazykov na riešenie problémov v rôznych vedeckých disciplínach. Popísaný špecifikačný jazyk patrí do skupiny moderných špecifikačných jazykov, o čom nás presvedčili jeho niektoré vlastnosti. Taktiež programovací funkcionálny jazyk bol vhodne vybraný pre jeho silnú typovosť. Tieto jazyky sú pomerne jednoduché, ale zároveň komplexné pre riešenie nášho problému. Ďalej bolo matematicky podložené vytvorenie inštitúcií špecifikácie požiadaviek ako aj špecifikácie návrhu, čo bolo reprezentované na jednoduchých príkladoch.

V danej téme sa dá pokračovať podrobnejším popisom jazykov a hlbším matematickým rozborom inštitúcií, čo prevyšuje rozmer tejto práce. Táto práca sa dá využiť v budúcnosti ako podklad pri študovaní životného vývoja softvéru od špecifikácie požiadaviek až po výslednú špecifikáciu a to asi v najdôležitejšej časti, prechodu od formálnej špecifikácie požiadaviek k formálnej špecifikácii návrhu, kedy sa formálne zapísané matematické podmienky zo špecifikačného jazyka zobrazia do návrhového jazyka. Taktiež na uvedených príkladoch je možné získať určitú predstavu o zobrazení z praktického hľadiska.

Táto práca prispela svojím obsahom k procesu vývoja korektného programovania.

Zoznam použitej literatúry

[1] P. Hudak a J. Peterson a J. H. Fasel. A gentle introduction to Haskell. Technical Report, Yale University, jún 2000.

[2] Valerie Novitzká a Ján Kollár. From Requirements Specification to Design Specification. *Journal of Information, Control and Management Systems*, 1(2):55–64, 2003.

[3] Simon Peyton Jones (editor). Haskell language and library: The Revised Report. Technical Report, Yale University, december 2002.

[4] Simon Peyton Jones. About Haskell. [online] Publikované, december 2001. Dostupné z <`http://www.haskell.org/aboutHaskell.html`>.

[5] Martin Kitka. Proces vývoja programov na základe formálnych špecifikácií. diplomová práca, Technická univerzita v Košiciach, Katedra počítačov a informatiky, 2004.

[6] V. Novitzká. *O teórii korektného programovania*. 1. vyd. Elfa, Košice, 2003. 117 s. ISBN 80-89066-61-5.

[7] The CoFI Task Group on Language Design. CASL - The CoFI Algebraic Specification Language – Language Summary. [online] Publikované, apríl 1997. Dostupné z <`http://www.brics.dk/Projects/CoFI/Documents/LanguageSummary/`>.

[8] The CoFI Task Group on Language Design. CASL - The CoFI Algebraic Specification Language Rationale. [online] Publikované, máj 1997. Dostupné z <`http://www.brics.dk/Projects/CoFI/Documents/CASL/Rationale/`>.

Zoznam príloh

1. CD médium – diplomová práca v elektronickej podobe

Zoznam obrázkov

2 – 1 relácia splnenia formúl 6

Zoznam tabuliek

4 – 1 systémy algebraických špecifikácií 24