Syntaxe a sémantika programovacích jazyků
Teorie programovacích jazyků

Filip Goldefus [1]

podzim 2007

[1]Vysoké učení technické Brno

Obsah

Kapitola 1

Úvod

Při návrhu a tvorbě programovacích jazyků se v 60. letech nepoužívaly formální prostředky k jejich návrhu a analýze. Jak se ale později ukázalo tak důsledná formální specifikace programovacího jazyka nejen upozorní na chyby v návrhu, ale také zpřehlední vytvořený kód a vede ke specifikaci formálního modelu který je znovupoužitelný.

K popisu struktury (syntaxe) programovacího jazyka se používá mnoha formalismů, v následujícím textu budou uvedeny především bezkontextové gramatiky a regulární výrazy.

Sémantika se zabývá významem (interpretací) programovacího jazyka, jenž lze popsat například denotační sémantikou, algebraickou sémantikou, sémantikou akcí, apod.

Tento dokument pojednává o formálním zápisu vybraných typů sémantických formalismů a syntaxí definovanou regulárním výrazem a bezkontextovou gramatikou ([1]).

Kapitola 2

Syntaxe

Syntaxe se zabývá upořádáním slov (lexikografických jednotek) jazyka ve větách. Každý jazyk obsahuje slovník symbolů, slov a také pravidla pomocí nichž můžeme skládat jednotlivé věty.

2.1 Bezkontextové gramatiky

2.1.1 Slova

Při tvorbě vět jazyka se používají symboly a slova. Nechť S je libovolná množina symbolů, pak řetězec (slovo) je konečná sekvence symbolů. Délka řetězce je určena počtem symbolů v řetězci. Prázdný řetězec, který označujeme ε má délku 0. Pro řetězce délky n nad množinou symbolů S bude použita notace S^n, kde $n \geq 0$. Především $S^0 = \{\varepsilon\}$. Dále se bude užívat notace S^* pro množinu všech konečných slov nad množinou symbolů S pro kterou platí

$$S^* = S^0 \cup S^1 \cup S^2 \cup \ldots$$

a S^+ je množina všech neprázdných řetězců nad množinou symbolů S, tedy

$$S^+ = S^1 \cup S^2 \cup \ldots = S^* - \{\varepsilon\}.$$

Příklad: Nechť množina $Morse$ obsahuje symboly $\bullet$ a $-$, pak

$$\begin{aligned} Morse^0 &= \{\varepsilon\} \\ Morse^1 &= \{\bullet, -\} \\ Morse^2 &= \{\bullet\bullet, \bullet-, -\bullet, --\} \\ Morse^* &= \{\varepsilon, \bullet, -, \bullet\bullet, \bullet-, -\bullet, --, \ldots\} \end{aligned}$$

2.1.2 Bezkontextové gramatiky

Za účelem definování syntaxe vybraného jazyka je nutné zavést následující notaci

- *terminální symboly (terminály)* – symboly z nichž se skládají slova jenž obsahují věty námi požadovaného jazyka,
- *neterminální symboly (neterminály)* – symboly z nichž se také skládají slova, ale pomocí těchto slov nelze vytvořit věty jazyka. V přirozených jazycích neterminální symboly reprezentují určitý slovní druh např. podmět, přísudek, ...,
- *startující symbol* – je vybraný neterminál (počáteční) z něj se postupně vytvářejí věty jazyka,
- *pravidla* – specifikují jak jsou věty vytvořeny z terminálních symbolů a jiných vět.

Příklad: Typická věta v anglickém jazyce je složena z těchto větných druhů (neterminálů):

$$Subject\ Object\ Noun\ Verb$$

a terminálů:

a am cat I is me mat rat see sees the .

následující pravidla určují jak jsou věty (*Sentence*) složeny z terminálů a neterminálů:

$$\begin{aligned}
Sentence &::= Subject\ Verb\ Object.\\
\\
Subject &::= \textbf{I}\\
Subject &::= \textbf{a}\ Noun\\
Subject &::= \textbf{the}\ Noun\\
\\
Object &::= \textbf{me}\\
Object &::= \textbf{a}\ Noun\\
Object &::= \textbf{the}\ Noun\\
\\
Noun &::= \textbf{cat}\\
Noun &::= \textbf{mat}\\
Noun &::= \textbf{rat}
\end{aligned}$$

$$
\begin{aligned}
Verb &::= \textbf{am} \\
Verb &::= \textbf{is} \\
Verb &::= \textbf{see} \\
Verb &::= \textbf{sees}
\end{aligned}
$$

Kde znak ::= lze interpretovat jako 'může být složeno z'. Takže například první pravidlo říká:

Věta může být složena z podmětu, následována slovesem, následována předmětem a je vždy ukončena terminálem '.'.

□

Nyní upřesníme použitý formalismus použitý v předchozí části. *Bezkontextová gramatika* je čtveřice:

$$G = (T, N, S, P),$$

kde T je konečná množina terminálních symbolů, N je konečná množina neterminálních symbolů, S je počáteční neterminál a P je konečná množina pravidel. Požadujeme, aby množiny N a T byly disjunktní ($N \cap T = \emptyset$) a $S \in N$. Každé pravidlo z P je tvaru $X ::= \alpha$, kde $X \in N$ a $\alpha \in (N \cup T)^*$ je řetězec terminálních a neterminálních symbolů.

Předpokládejme že $X ::= \alpha$, $X ::= \beta$ a $X ::= \gamma$ jsou pravidla z N. Pak se častěji používá následující notace:

$$X ::= \alpha|\beta|\gamma$$

k seskupení těchto pravidel. Zde symbol '|' reprezentuje 'nebo'. Například pravidlo pro předmět z předchozího přikladu lze přepsat následovně:

$$
\begin{aligned}
Subject &::= \textbf{I} \\
&\mid \textbf{a}\ Noun \\
&\mid \textbf{the}\ Noun
\end{aligned}
$$

2.1.3 Syntaktický strom

Z definice gramatiky je zřejmé, že volba pravidel při tvorbě vět jazyka není určena. Vzhledem k této vlastnosti lze obecně některé věty tvořit použitím pravidel v různém pořadí.

Pokud existuje v jazyce, který je definován bezkontextovou gramatikou, alespoň jedna věta, která lze vytvořit aplikováním pravidel v různém pořadí, pak je gramatika *víceznačná*.

K zachycení struktury tvorby vět lze použít *syntaktické stromy*, které jsou definovány následovně, mějme gramatiku G pak:

- Pro libovolný neterminál X v G, X-strom má kořenový uzel označený X, jeho podstromy jsou X_1-strom, X_2-strom, $\ldots X_n$-strom, právě tehdy pokud $X ::= X_1 X_2 \ldots X_n$ je pravidlo v gramatice G. Pokud je pravidlo tvaru $X ::= \varepsilon$ pak X-strom nemá žádné podstromy.
- Pro libovolný terminál t v G je t-strom reprezentován jediným uzlem označeným hodnotou t.

Za povšimnutí stojí, že neterminál X se obecně může vyskytovat na levé straně mnoha pravidel v gramatice, vzhledem k tomu může existovat také více metod jak zkonstruovat N-strom.

2.1.4 Fráze, slova a jazyky

Fráze v gramatice G představuje řetězec terminálních symbolů složených z koncových uzlů (listů) syntaktického stromu. Přesněji, pro každý neterminál X z G, X-fráze v gramatice G je řetězec terminálních symbolů označujících koncové uzly syntaktického stromu s kořenem označeným X. Slovo gramatiky G je S-fráze, kde S je počáteční neterminál G. Jazyk generovaný gramatikou G je množina všech S-frází gramatiky G.

2.1.5 Struktura frází a sémantika

Je důležité poznamenat, že definice jazyka je velmi přímočará. Definuje jazyk jako množinu řetězců symbolů. Neříká ale nic o sémantice vytvořených frází. Kdykoliv se setkáme s pojmem jazyka, musíme mít na zřeteli je-li jazyk použit v přímočarém smyslu jako množina slov nebo v širším smyslu jako množina slov a s jejich významy (sémantikou).

Bezkontextové gramatiky nenabízí pouhé generování slov jazyka, ale jsou schopny určit frázovou strukturu každé věty včetně jejich syntaktických stromů.

Tuto vlastnost lze využít k určení významu věty (fráze) aplikováním nějaké interpretace syntaktickému stromu.

2.1.6 Abstraktní syntaxe

V předchozí části lze vidět, že změny v gramatice libovolného jazyka mohou změnit frázovou strukturu jejich vět. Proto se musí při návrhu jazyka klást důraz na tvorbu gramatiky a neopomenout při tom např. prioritu, asociativitu operátorů či víceznačnost syntaktických stromů. V počátcích návrhu programovacího jazyka je celková struktura jazyka a jeho sémantika velmi významná. Konkrétní struktura vět jazyka není v tomto stádiu nejdůležitější částí návrhu.

Ukázalo se výhodné vytvořit rozdíl mezi *abstraktní syntaxí* a *konkrétní syntaxí*. Abstraktní syntaxe se zaměřuje pouze na hierarchické vztahy mezi frázemi a podfrázemi např. že příkaz **if** je složen z jednoho výrazu a dvou příkazů. Konkrétní syntaxe se zaměřuje nejen na hierarchickou strukturu frází, ale také na konkrétní symboly pužívané k oddělování, uzávorkování apod.

2.1.7 Vlastnost sebevložení

Bezkontextová gramatika G má vlastnost sebevložení, pokud pro neterminál X z G existuje derivace $X \Rightarrow^+ \alpha X \beta$ a platí, že $\alpha \neq \varepsilon$ a $\beta \neq \varepsilon$. Pokud je jazyk bezkontextový a není regulární, pak bezkontextová gramatika má vždy tuto vlastnost.

2.2 Regulární výrazy

Vlastnost sebevložení je pro definici programovacích jazyků nutná, ale důležitá podtřída bezkontextových jazyků jenž je definována gramatikami bez vlastnosti sebevložení, tyto gramatiky se nazývají *regulární*.

2.2.1 Definice regulárních výrazů

Nechť S je libovolná množina symbolů. Regulární výraz (RV) nad množinou S je formule definována následující tabulkou.

RV E	Množina slov definovaná RV $[[E]]$	Popis
ε	$\{\varepsilon\}$	prázdné slovo
s	$\{s\}$	terminální symbol
$F \cdot G$	$\{\alpha\beta \mid \alpha \in [[F]], \beta \in [[G]]\}$	zřetězení
$F + G$	$[[F]] \cup [[G]]$	sjednocení
F^*	$\{\alpha_1\alpha_2 \ldots \alpha_n \mid n \geq 0; \alpha_1, \ldots, \alpha_n \in [[F]]\}$	iterace
(F)	$[[F]]$	

kde F a G jsou libovolné regulární výrazy a s je libovolný terminální symbol z S.

2.3 Syntaxe programovacího jazyka

V této sekci budou představeny techniky využívající zavedené pojmy ke specifikaci programovacích jazyků. Jako příklad bude sloužit programovací jazyk Δ, jenž vychází z programovacího jazyka Pascal.

2.3.1 Mikrosyntaxe

Ačkoliv je program reprezentován textem složeným z jednotlivých znaků, je vhodné rozlišovat tzv. lexikografické jednotky což jsou např. literály, identifikátory, operátory, apod. Mikrosyntaxe definuje právě tyto entity.

V programovacím jazyce Δ (a ve většině ostatních programovacích jazycích) obsahuje zdrojový kód komentáře a bílé znaky. Jejich obsah a pozice v textu nemají žádný vliv na frázovou strukturu. Vzhledem k tomu mikrosyntaxe definuje i tyto elementy.

Část mikrosyntaxe jazyka Δ je uvedena v následující tabulce.

Makrosyntaxe:	Program	::=	*Command*
	Command	::=	**let** Declaration **in** Command |
			V-name **:=** Expression | ...
	Expression	::=	Literal | V-name | Operator ... | ...
	V-name	::=	Identifier | ...
	Declaration	::=	**var** Identifier **:** Type-denoter | ...
	...		
Mikrosyntaxe:	Program	::=	(Token | Comment| Blank)*
	Token	::=	Integer-Literal | Char-Literal |
			Identifier | Operator |**in**|**let**|**var**|
			:|**:=**|...
	Integer-Literal	::=	...
	Char-Literal	::=	...
	Identifier	::=	...
	Operator	::=	...
	Comment	::=	...
	...		

2.3.2 Makrosyntaxe

Dalším úkolem při návrhu programovacího jazyka je specifikace struktury celého programu. Pro tyto účely se používá makrosyntaxe. Část makrosyntaxe programovacího jazyka Δ je uvedena v předchozí tabulce.

Makrosyntaxe je obvykle sepcifikována bezkontextovou gramatikou. Množina neterminálních symbolů obsahuje tyto elementy **Program** (počáteční neterminál tvořící třídu všech programů), **Command**, **Expression** a **Declaration**. Terminální symboly jsou entity definované mikrosyntaxí což představují neterminály **Integer-Literal**, **Character-Literal**, **Identifier** a **Operator**.

V programovacím jazyce Δ existují tři různé typy priority operátorů: unární operátory mají vyšší prioritu než binární operátory. Nejnižší úroveň priority (Expression) mají **if-** a **let-** výrazy. Střední úroveň priority (secondary-Expression) zahrnuje binární operátory (všechny binární operátory mají stejnou prioritu). Nejvyšší úroveň priority (primary-Expression) mají pouze unární operátory.

Kapitola 3

Denotační sémantika

Denotační sémantika byla vytvořena na počátku 70. let Christopherem Stracheyem a Danou Scottovou. Dřívější snahy o formalizování sémantiky byly spíše zaměřeny na operační význam programů. Narozdíl od těchto snah se Ch. Strachey a D. Scottová zaměřili na vyjádření sémantiky programovacího jazyka čistě na matematickém základu (tato sémantika se dříve nazývala matematická).

Denotační sémantika přinesla mnoho výhod. Mimo jiné můžeme předpovídat chování programu bez toho aniž byhom jej spuštěli na počítači. Další výhodou denotační sémantiky je, že se můžeme vyjadřovat o vlastnostech programů např. ekvivalence dvou programů.

3.1 Sémantické funkce

V denotační sémantice se význam každé fráze – každý výraz, příkaz, deklarace, ale také celý program – reprezentuje vhodnou matematickou entitou. Tato entita se nazývá *denotace* fráze. Sémantiku programovacího jazyka určíme tak, že vytvoříme funkce které přiřadí frázím jejich denotace. Použité funkce se pak nazývají *sémantické funkce*.

Základní myšlenky denotační sémantiky:

- Význam každé fráze p bude určen hodnotou d patřící do nějaké domény. Hodnotu d budeme nazývat denotace fráze p. Nebo také fráze p denotuje hodnotu d.
- Pro každou množinu frází P určíme doménu D jejich denotací a vytvoříme sémantickou funkci f, která zobrazuje každou frázi z P na jeji denotaci z množiny D. Tuto funkci budeme zapisovat $f : P \to D$.
- Sémantickou funkci f definujeme pomocí množiny sémantických rovnic, jedna pro každý různý tvar fráze v P. Pokud např. fráze P obsahuje Q a R jako podfráze, pak korespondující sémantická rovnice bude vypadat následovně:

$$f[[\ldots Q \ldots R \ldots]] = \ldots f'Q \ldots f''R \ldots$$

kde f' a f'' jsou sémantické funkce vhodné pro Q a R. Jinak řečeno, denotace každé fráze je definována v kontextu denotací jeho podfrází.

3.1.1 Notace pro definici funkcí

Pro definici sémantických funkcí a pomocných funkcí můžeme použít jenoduchou podmnožinu běžného matematického zápisu. Pro zavedení nových proměnných budeme používat notaci $'\mathbf{let} \ldots \mathbf{in} \ldots'$, např.:

$$\mathbf{let}\ s = 0.5 \times (a + b + c)\ \mathbf{in}\ sqrt(s \times (s - a) \times (s - b) \times (s - c))$$

Notace **let** může také zavádět nové funkce:

$$\mathbf{let}\ succ\, n = n + 1\ \mathbf{in}\ \ldots succ\, m \ldots succ(succ\, i) \ldots$$

Často bude užitečné definovat anonymní funkce. Např.

$$\lambda n.n + 1$$

označuje funkci která přiřazuje hodnotě n hodnotu $n + 1$. (Jinak lze funkci také označit jako $'n \to n + 1'$). Předchozí definice funkce *succ* je zkratkou zápisu:

$$\mathbf{let}\ succ\, n = \lambda n.n + 1\ \mathbf{in}\ \ldots$$

3.2 Domény

V této sekci si představíme základní a složené domény, které bývají častěji označovány jako datové typy.

3.2.1 Základní datové typy

Základní datové typy obsahují primitivní elementy, jenž nejsou dále rozložitelné na jednodušší datové typy. V běžných programovacích jazycích se vyskytují tyto datové typy:

- **Character** – elementy jsou prvky nějaké abecedy znaků.
- **Integer** – elementy jsou pozitivní a negativní prvky množiny celých čísel (včetně nuly).
- **Natural** – elementy jsou nezáporná celá čísla.
- **Truth-Value** – elementy jsou prvdivostní hodnoty *truth* nebo *false*.
- **Unit** – jediným prvkem je () – prázdná n-tice.

3.2.2 Odvozené datové typy

Pomocí operací, které budou uvedeny v následujících odstavcích lze vytvořit složitější datové typy užitím primitivních datových typů.

Kartézský součin

Datový typ kartézský součin obsahuje elementy, které jsou tvořeny uspořádanými n-ticemi. Každý prvek kartézského součinu nad typem $D \times D'$ je uspořádaná dvojice (x, x') taková, že $x \in D$ a $x' \in D'$. Tato definice může být rozšířena na obecnou n-tici $D_1 \times \ldots \times D_n$ a prvek tohoto datového typu je uspořádaná n-tice $(x_1, x_2, \ldots, x_n)$.

Dále budeme pužívat značení $(\ldots, \ldots)$ pro konstrukci n-tic. Pro kompozici i dekompozici n-tice lze použít příkaz **let ... in ...**. Např.

$$\textbf{let } (amount, denom) = pay \textbf{ in} \ldots.$$

Disjunktní sjednocení

Datový typ vzniklý operací disjunktní sjednocení obsahuje elementy, jenž jsou prvky prvního nebo druhého primitivního datového typu (ne obou současně, množiny musí být disjunktní). Každý prvek disjunktního sjednocení datových typů $D + D'$ je buď $left\ x$ což znamená $x \in D$ nebo $right\ x'$ a platí $x' \in D'$.

Například datový typ $Shape$ může být definován následovně:

$$Shape = rectangle(Real \times Real) + circle\ Real + point$$

Funkce primitivních typů

Doména funkce obsahuje prvky jenž jsou definovány jako zobrazení. Každý prvek domény $D \to D'$ je funkce, která zobrazuje prvky z množiny D na prvky z množiny D'

3.3 Prostředí

Deklarace v programovacím jazyce vytvářejí vazby mezi identifikátory a entitami. Každá vazba má určitou platnost v rámci programu. Typicky je deklarace platná v bloku, kde byla vytvořena.

V následujícím příkladu je proměnná n platná na řádcích 1-5 a proměnná m je platná na řádcích 3-5.

```
1 let val m = 10
2 in
3  let val n = m * m
4  in
5    m+n
```

Uvažujeme-li část programu např. výraz nebo příkaz, pak můžeme říct, že tato část je vyhodnocena nebo spuštěna v určitém *prostředí*, které je definováno množinou všech přiřazení platných v dané části programu.

3.3.1 Modely prostředí

Vlastnosti prostředí jsou poměrně nezávislé na programovacím jazyce. V dalším textu budeme doménu identifikátorů označovat **Identifier**, doménu přiřazení označovat **Bindable** a doménu prostředí **Environ**. Možnosti charakterizace prostředí budou specifikovány následujícími funkcemi:

$empty - environ$	**: Environ**		
$bind$	**: Identifier** $\times$ **Bindable**	$\rightarrow$	**Environ**
$overlay$	**: Environ** $\times$ **Environ**	$\rightarrow$	**Environ**
$find$	**: Environ** $\times$ **Identifier**	$\rightarrow$	**Bindable**

Tyto funkce můžeme neformálne definovat následovně:

- *empty-environ* – vrací prázdné prostředí, které neobsahuje žádné přiřazení – $\{\}$.
- *bind(I, bdble)* – vrací prostředí obsahující jediné přiřazení ve kterém je identifikátoru I přiřazeno přiřazení *bdble*.
- *overlay(env', env)* – vrací prostředí kombinující přiřazení env a env', pokud se některý z identifikátorů vyskytuje v obou prostředích pak se použije pouze přiřazení z prostředí env'.
- *find(env, I)* – vrací přiřazení, které je přiřazeno k proměnné I v prostředí env, pokud proměnná přiřazení nemá je vrácena hodnota $fail$.

3.4 Sémantika jazyků s přiřazením

S využitím funkcí definovaných v předchozích odstavcích můžeme specifikovat sémantiku jazyků s přiřazeními. Definujme například jazyk EXP následující abstraktní syntaxí:

$Expression$	::=	$Numeral$
	|	$Expression + Expression$
	|	$\ldots$
	|	$Identifier$
	|	**let** $Declaration$ **in** $Expression$
$Declaration$	::=	**val** $Identifier = Expression$

V jazyce EXP jsou pouze přirozená čísla přiřaditelná –

$$Bindable = Integer.$$

V jazyce EXP jsou výsledky pouze přirozená čísla. Pokud uvažujeme například výraz $'m * m'$, pak zjistíme že hodnota tohoto výrazu je závislá na prostředí

ve kterém se daný výraz vyskytuje. Nechť tedy význam (denotace) výrazu je zobrazení z množiny prostředí na množinu přirozených čísel

$$evaluate : Expression \to (Environ \to Integer).$$

Například $evaluate[[m * m]]$ přiřadí prostředí $\{m \to 2, \ldots\}$ hodnotu 4, prostředí $\{m \to 3, \ldots\}$ hodnotu 9, ...

Výsledek zpracování deklarace také závisí na prostředí, ale výsledkem není pouze přirozené číslo, ale množina přiřazení – prostředí. Nechť tedy význam (denotace) deklarace je zobrazení z množiny prostředí na prostředí

$$elaborate : Declaration \to (Environ \to Environ).$$

Například $elaborate[[\textbf{val}\ n = m * m]]$ přiřadí prostředí tvaru $\{m \to 1, \ldots\}$ množinu $\{n \to 1\}$, $\{m \to 2, \ldots\}$ množinu $\{n \to 4\}$, ...

Sémantické rovnice pro výrazy zapisujeme ve tvaru:

$$evaluate\ [[E]]\ env = \ldots,$$

kde pravá strana je výraz který obdržíme vyhodnocením E v prostředí env. Sémantické rovnice pro výrazy jsou vyjádřeny následovně,

$$evaluate\ [[N]]\ env = valuation\ N$$

výsledek vyhodnocení výrazu obsahujícího číslo z N v prostředí env, je hodnota tohoto čísla.

$$evaluate\ [[E_1 + E_2]]\ env = sum(evaluate\ E_1\ env, evaluate\ E_2\ env)$$

výsledek vyhodnocení výrazu tvaru $'E_1 + E_2'$, v env je součet výsledků vzniklých vyhodnocením výrazů E_1 a E_2 v prostředí env. Ostatní aritmetické jsou definovány obdobně.

$$evaluate\ [[I]]\ env = find(env, I)$$

výsledek vyhodnocení výrazu, který vznikl nahrazením výskytu proměnné I v prostředí env, hodnotou kterou má proměnná v prostředí env.

$$\begin{aligned} evaluate[[\textbf{let}\ D\ \textbf{in}\ E]]\ env \quad = \quad & \textbf{let}\ env' = elaborate\ D\ env\ \textbf{in} \\ & evaluate\ E(overlay(env', env)) \end{aligned}$$

výsledek vzhodnocení výrazu tvaru $'\textbf{let}\ D\ \textbf{in}\ E'$ v prostředí env je výsledek vyhodnocení výrazu E v novém prostředí vzniklém z prostředí env přiřazením hodnoty D do nového prostředí env'.

Sémantické rovnice používané pro deklarace se zapisují ve tvaru

$$elaborate\ [[D]]\ env = \ldots,$$

kde pravá strana je výraz vytvářející přiřazení vytvořená vyhodnocením D v prostředí env. V jazyce EXP je možný jediný tvar deklarací a sémantická rovnice vypadá následovně

$$elaborate[[\textbf{val}\ I = E]]\ env = bind(I, evaluate\ E\ env)$$

výsledek vyhodnocení deklarace tvaru $'\textbf{val}\ I = E'$ v prostředí env, je přiřazení hodnoty výrazu E proměnné I.

Kapitola 4

Algebraická sémantika

V této kapitole se budeme věnovat specifikací abstraktních typů. Abstraktní typ je charakterizován množinou operací a také konstantami. Hodnoty daného typu jsou vyjádřeny nepřímo, jsou vytvořeny opakovaným použitím operací na konstantách daného typu. Často lze některé hodnoty vytvořit více způsoby, existují tudíž axiomy vyjadřující vztah mezi některými operátory.

Jelikož množina hodnot společně s množinou operací tvoří algebru, je přirozené dát význam abstraktnímu typu užitím algebry. Z toho důvodu se užitá metoda nazývá algebraická sémantika.

4.1 Základní koncepty

U algebraické sémantiky je vhodné uvažovat následující tři koncepty: *specifikaci* abstraktního typu, *teorii* udávanou specifikací a *algebry*, které splňují teorii.

Specifikace (nebo přesněji *algebraická specifikace*) se skládá ze dvou částí: popisu a axiomů. Popis definuje druhy (typy), které budou specifikovány, symboly operátorů. Axiomy jsou logické výrazy popisující chování operací (v této kapitole axiomy jsou vždy rovnice). Protože uvažujeme abstraktní typy, nebudeme definovat reprezentaci hodnot. Místo toho použijeme axiomy pro vyjádření vztahů mezi operátory.

Jako příklad můžeme uvažovat specifikaci abstraktního typu pravdivostních hodnot. Specifikace tohoto typu musí obsahovat konstanty $true$ a $false$ (pro vyjření pravdy a nepravdy) a další operace $'not'$ (negace), $'\wedge'$ (konjukce), $'\vee'$ (disjunkce) a $'\Rightarrow'$ (implikace).

specification TRUTH-VALUES

sort Truth-Value

operations
true : Truth-Value

true : Truth-Value
not_ : Truth-Value → Truth-Value
_∧_ : Truth-Value, Truth-Value → Truth-Value
_∨_ : Truth-Value, Truth-Value → Truth-Value
_⇒_ : Truth-Value, Truth-Value → Truth-Value

variables t, u: Truth-Value

equations

not true	= false
not false	= true
t ∧ true	= t
t ∧ false	= false
t ∧ u	= t ∧ u
t ∨ true	= true
t ∨ false	= t
t ∨ u	= t ∨ u
t ⇒ u	= (not t)∨ u

end specification

Specifikace je uvedena speciálním klíčovým slovem **specification** následována názvem specifikovaného typu, v předchozím případě tedy TRUTH-VALUES. Konec specifikace je označen klíčovými slovy **end specification**. Záhlaví je následováno slovem **sort(s)**, které uvádí použité typy. Další klíčové slovo **operation(s)** zavádí množinu operátorů, společně s typy jejich vstupních a výstupních hodnot. Pozice proměnné každého operandu je indikováná speciálním symbolem _. Absence tohoto symbolů u operací true a false znamená, že tyto operace jsou vlastně konstantami (nulární operátory). Pozice znaku _ v operaci *not* _ určuje, že operátor not je unární a prefixový. Pozice znaků _ v operacích _∧_, _∨_ a _⇒_ určuje, že tyto operátory jsou binární a infixové.

Pro každý typ existuje množina tzv. dobře utvořených *termů*. Neformálně jsou tyto termy vytvořeny následovně:

- každá konstanta je term,
- každá proměnná je term,
- aplikace operátoru na vhodný počet a typ termů je term.

Například následující termy jsou typu Truth-Value:

$true$	$false$	
t	u	
$not\ true$	$true \vee false$	$true \Rightarrow false$
$t \vee true$	$t \vee u$	$(t \vee u) \wedge u$

Ale například $' \Rightarrow false'$ není term, neboť ⇒ je binární operátor, a např. $'true \vee 0'$ také není term protože 0 není typu Truth-Value.

Termy které neobsahují proměnné se nazývají atomické termy (např. $'true'$, $true \vee false, \ldots$). Například v předchozí tabulce jsou atomické termy na prvním a třetím řádku. Atomický term může vzniknout z termu obsahujícího proměnné substitucí konstant za všechny proměnné v daném termu.

V poslední části specifikace se objevuje klíčové slovo **equation(s)**, které obsahuje množinu axiomů (rovnic). Axiomy se dělí na *podmíněné* a *nepodmíněné*. Nepodmíněné rovnice obsahují pouze vztahy mezi termy stejného typu. Rovnice (axiomy) ve specifikaci TRUTH-VALUES jsou pouze nepodmíněné.

Proměnné a jejich typy, které se vyskytují v rovnicích, jsou uvedeny za klíčovým slovem **variable(s)**.

Rovnice dávají význam termům, například rovnice $t \wedge true = t$ může být neformálně interpretován jako: " pro libovolnou pravdivostní hodnotu t, je hodnota $t \wedge true$ rovna hodnotě t". Tuto rovnici můžeme vyjádřit více formálně pomocí atomických termů a substitucí.

Atomická rovnice je rovnice mezi atomickými termy. Např. $not\ true = false$ je atomická rovnice. Atomické rovnice mohou být vytvořeny z jiných rovnic substitucí proměnných za atomické termy. Např. term $t \wedge true = t$ vytváří náseldující atomické rovnice:

$true \wedge true$	$=$	$true$	substitucí $true$ za t
$false \wedge true$	$=$	$false$	substitucí $false$ za t
$(not\ true) \wedge true$	$=$	$not\ true$	substitucí $'(not\ true)'$ za t
atd.			

Každá z těchto rovnic vznikne substitucí atomického termu typu Truth-Value za oba výskyty proměnné t v rovnici $t \wedge true = t$. Obecně všechny výskyty libovolné proměnné musí být substituovány stejným atomickým termem, který musí být stejného typu jako proměnná. Dohromady tvoří axiomy logickou *teorii*.

Neformálně, *algebra* je tvořena množinou hodnot (někdy nazývanou nosič), dohromady s konstantami a funkcemi nad množinou hodnot. Například množina přirozených čísel $\{0, 1, 2, 3, \ldots\}$ společně s operacemi '+' (součet) a '*' (součin) tvoří algebru.

Algebra splňuje teorii pokud všechny rovnice v teorii jsou splněny v algebře, po transformování operátorů teorie na operátory algebry (také se občas uvádí, že algebra je model dané teorie). Uvažujme například algebru bitů s nosičem $\{0, 1\}$ a následujícími funkcemi:

$flip$	**where**	$flip(0) = 1,$	$flip(1) = 0$		
$*$	**where**	$0 * 0 = 0,$	$0 * 1 = 0$	$1 * 0 = 0$	$1 * 1 = 1$
$+$	**where**	$0 * 0 = 0,$	$0 + 1 = 1$	$1 + 0 = 0$	$1 + 1 = 1$
$\leq$	**where**	$0 \leq 0 = 1,$	$0 \leq 1 = 1$	$1 \leq 0 = 0$	$1 \leq 1 = 1$

Tato algebra splňuje teorii generovanou specifikací TRUTH-VALUES. Specifikace a algebra mají obdobné hodnoty. Operace jsou prevedeny následovně.

$false$	corresponds to	0
$true$	corresponds to	1
not	corresponds to	$flip$
$\vee$	corresponds to	$+$
$\wedge$	corresponds to	$*$
$\Rightarrow$	corresponds to	$\leq$

Například uvažujme rovnici $t \wedge true = t$. V algebře je tato rovnice přeložena na $t * 1 = t$ a je splněna neboť $0 * 1 = 1$ a $1 * 1 = 1$.

Obecně může specifikace zavést více typů, tudíž musíme zobecnit naši definici algebry. *Vícetypová algebra* sestává z systému nosičů společně s konstantami a funkcemi nad těmito nosiči.

4.1.1 Specifikace uspořádané dvojice

V této sekci uvedeme příklad specifikace uspořádané dvojice. Každá uspořádaná dvojice se skládá ze dvou komponent. Uspořádané dvojice jsou charakterizovány následujícími operacemi: vytvoření dvojice, výběr prvního prvku dvojice, výběr druhého prvku dvojice. Obecně prvky uspořádané dvojice mohou být různého typu, ale pro jednoduchost budeme uvažovat oba prvky stejného typu – **Component**. Specifikace je následující:

specification ORDERED-PAIRS

formal sort Component
sort Pair

operations

pair(_, _)	: Component, Component → Pair
first field of _	: Pair → Component
second field of _	: Pair → Component

variables c, c': Component

equations

first field of pair(c, c')	= c
second field of pair(c, c')	= c'

end specification

Component je specifikována jako formální typ. Tato specifikace umožňuje instanciovat ORDERED-PAIRS na uspořádané dvojice přirozených čísel, pravdivostních hodnot, atd.

Instanciace této specifikce užitím TRUTH-VALUES,

specification TRUTH-VALUE-PAIRS

include instantiation of ORDERED-PAIRS **by** TRUTH-VALUES
using Truth-Value **for** Component

end specification.

Například dvojice $pair\ (false, true)$ je typu $Pair$.

Kapitola 5

Závěr

Teorie zabývající se programovacími jazyky je velmi obsáhlá, pro její pečlivé studování je nezbytné čerpat mnoho zkušeností z mnoha vědních oborů jako je matematika, teorie jazyků, lingvistika, apod. jenž jsou její nedílnou součástí. Tato esej se nesnaží objasnit všechny aspekty návrhu programovacího jazyka, ale pouze naznačit zajímavé oblasti této teorie. Čtenář se může pro bližší a podrobnější informace poohlédnout po literatuře věnující se tomuto tématu.

Literatura

[1] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International, 1991.