FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Sémantika programovacích jazyků

Semestrální práce v předmětu Teorie programovacích jazyků

Pavel Cagaš

2003

Obsah

1 Úvod

Formální metody popisu syntaxe se poměrně záhy po vzniku prvních překladačů staly neodmyslitelnou součástí specifikace každého nového programovacího jazyka. Také se jich začalo s velkým úspěchem používat při implementaci syntaktických analyzátorů překladačů. Zatímco vývoj prvních překladačů jazyků bez formální specifikace (FORTRAN) byl zdlouhavý a pracný úkol, aplikace matematického formalismu (specifikace gramatiky jazyka) umožní použít obecně známých technologií (např. syntaxí řízený překlad) k výraznému zkrácení doby vývoje, redukci složitosti překladače, zlepšení jeho udržovatelnosti, omezení chyb a usnadnění ladění.

1.1 Formální specifikace syntaxe

Formální specifikace syntaxe programovacích jazyků má počátky ve druhé polovině 50. let minulého století, kdy nezávisle na sobě John Backus a Noam Chomsky přišli různými cestami k prakticky totožné notaci specifikace jazyka.

Chomsky byl lingvista a jeho výzkum gramatik jakožto generátorů jazyků byl motivován zájmem o přirozené jazyky. Přesto zejména dvě třídy jeho čtyřčlenného dělení (bezkontextové a regulární gramatiky a jimi generované jazyky) našly velkého uplatnění v informačních technologiích.

Backus poprvé specifikoval syntax programovacího jazyka Algol 58 s pomocí své notace, která se stala po malých modifikacích provedených Petrem Naurem všeobecně známá jako Backus-Naurova forma (BNF) specifikace.

Je až s podivem, že podstata BNF je prakticky totožná s Chomského bezkontextovými gramatikami. Oba formalismy lze používat téměř identicky.

1.2 Formální specifikace sémantiky

Různé způsoby formální specifikace sémantiky se objevují na přelomu 60. a 70. let minulého století. Floyd a Hoare pracovali na axiomatické sémantice, významová sémantika (denotational semantics) byla vyvinuta v 70. letech Christopherem Stracheym.

Přesto že formální specifikace sémantiky není o mnoho mladší než teorie gramatik, na formální specifikaci sémantiky jazyka lze narazit jen naprosto výjimečně. Sémantika programovacích jazyků bývá popisována neformálně (přirozeným jazykem) a to může vést k nejednoznačnostem na straně programátorů programovací jazyk užívajících i těch, co implementují jeho překladač.

Absence formální specifikace sémantiky se netýká jen programovacích jazyků, ale také moderních komponentových vývojových systémů (např. modelovací jazyk UML). Uživatelé komponent tak mají k dispozici přesnou specifikaci syntaxe rozhraní (názvy metod a jejich parametry) včetně popisu statické sémantiky rozhraní (datové typy parametrů), avšak sémantika rozhraní bývá opět popisována pouze běžným jazykem.

Neformální popis sémantiky je stále používán, protože dosud vyvinuté způsoby formálního popisu sémantiky jsou velice komplexí jejich praktická použitelnost je dosti sporná. Existují specifikace sémantiky některých programovacích jazyků, např. Algol 60 či Pascal. Dokonce existují experimentální systémy, které na základě formální specifikace sémantiky vygenerují překladač daného jazyka. Ovšem takový překladač je velmi neefektivní a v praxi

nepoužitelný. Formální specifikace sémantiky tak stále zůstávají doménou akademického výzkumu např. na poli dokazování správnosti algoritmů.

1.3 Křehké bázové třídy

Nedodržení syntaktické správnosti programu překladače zcela spolehlivě detekují a problémy způsobené syntaktickou analýzou jsou spíše nepodstatné. Často se jedná třeba jen o nepřesné hlášení chyb nebo problematické zotavení překladače, ale programátoři si brzy uvědomí že například chybové hlášení „očekáváno END“ může být způsobeno i chybějícím středníkem.

Moderní překladače přidávají některé sémantické kontroly, které mohou programátorům výrazně usnadnit život. Jsou schopny odhalit sémantickou chybu v syntakticky správných konstrukcích, jako např. „ne všechny cesty v implementaci funkce vrací hodnotu“ či „proměnná použita bez předchozí inicializace“.

Ovšem trend vývoje software kopíruje jiná odvětví (např. elektroniku – elektronická zařízení nejsou navrhována z diskrétních součástek, namísto toho jsou používány integrované obvody, tedy zapouzdřené komponenty s definovaným rozhraním). Programová komponenta je definována nejen syntaxí svého rozhraní, ale i jeho sémantikou. A zatímco dodržení syntaktické správnosti při volání rozhraní objektu je možné spolehlivě zajistit již ve fázi vývoje, změny sémantiky jsou prakticky neodhalitelné. Přes veškerou snahu programátorů sémantika komponent prostupuje stěnami „černé skříňky“ programové komponenty (objektu) a její (byť nechtěné) změny, třeba při nahrazení starší verze komponenty verzí novější, mohou mít pro běh programového systému velmi fatální následky. V objektově orientovaných jazycích je tento problém nazýván *problém křehkých bázových tříd*.

Použití automatické kontroly sémantických závislostí komponenty a upozornění na změny sémantiky, které mohou způsobit nefunkčnost programového systému by mohlo významně zvýšit robustnost a spolehlivost moderních komponentových aplikací.

2 Způsoby specifikace sémantiky

Dosud bylo popsáno několik způsobů více či méně formální specifikace sémantiky programovacího jazyka:

- **Operační sémantika** – popisuje význam algoritmu jeho vykonáním na skutečném či virtuálním (simulovaném) stroji. Význam příkazu programovacího jazyka je reprezentován změnou stavu stroje před a po vykonání příkazu.
- **Axiomatická sémantika** – byla vyvinuta jako součást snahy vyvinout metody dokazování správnosti algoritmů. Každý příkaz je předcházen a následován logickými výrazy, které omezují platnost proměnných. Význam příkazu tedy není definován změnou stavu celého stroje, ale jen daných proměnných.
- **Významová sémantika (denotational semantics)** - přiřazuje *význam* každé *syntaktické frázi* programovacího jazyka (deklarace, příkaz, …). Smysl každé fráze je určen matematickou entitou – *významem*. Ke každé frázi je zapotřebí specifikovat funkci, která jí přiřadí daný význam. Tato funkce je nazývána *sémantická funkce*.

2.1 Operační sémantika

Význam algoritmu je v operační sémantice definován změnou stavu skutečného či virtuálního stroje (počítače) před a po vykonání algoritmu. Protože instrukce, z nichž je algoritmus

složen, pracují nejen s operační pamětí, ale také s datovými registry, mění stavové a podmínkové registry apod., stav stroje musí mimo paměť zahrnovat i tyto registry.

Použití skutečného počítače ale přináší jisté komplikace. Současné počítače (procesory) jsou dosti komplexní a celou situaci dále komplikuje skutečnost, že programy nepracují přímo na daném hardware, ale využívají služeb operačního systému. Stav programu je tak ovlivňován i mechanizmy, které program samotný nedokáže ovlivnit – například stav synchronizačních objektů jádra operačního systému, který může být měněn systémem v závislosti na událostech zcela nesouvisejících s prováděným programem.

Definice sémantiky změnou stavu reálného počítače je navíc přístupné pouze pro lidi mající k dispozici zcela stejný počítač.

Tyto problémy mohou být eliminovány nahrazením skutečného počítače virtuálním strojem. Sémantika je pak definována změnou stavu tohoto virtuálního stroje, který teprve je emulován nějakým skutečným počítačem. To přináší značné výhody, protože virtuální stroj může být podstatně jednodušší a může být přímo navrhován s ohledem na definici sémantiky algoritmu. Navíc virtuální stroj může pracovat na různých skutečných počítačích.

K definici operační sémantiky virtuálním strojem je ale zapotřebí nejen stroj samotný (interpreter instrukcí), ale také překladač, který algoritmus zapsaný v daném jazyce do těchto instrukcí přeloží.

Koncept operační sémantiky není nijak výjimečný a je požíván např. v učebnicích programovacích jazyků, kdy složitější konstrukce (např. programové smyčky) jsou vysvětlovány s použitím jednodušších „pseudoinstrukcí“ (např. **if** podmínka **goto** návěští):

Příkaz smyčky:	*Význam definovaný s použitím pseudokódu:*
while condition **do**	loop:
…	**if not** condition **goto** exit
end	…
	goto loop
	exit:

Operační sémantika poskytuje praktický prostředek pro popis sémantiky jazyka pro uživatele daného jazyka i pro programátory tvořící překladač pro tento jazyk. Popis sémantiku je poměrně jednoduchý. Je to také popis neformální, protože operační sémantika závisí na algoritmech, nikoliv na matematice.

Příkazy jazyka jsou definovány pomocí příkazů jiného jazyka (instrukční sady reálného procesoru nebo virtuálního stroje). To může vést k definici v kruhu např. při pokusu popsat sémantiku virtuálního stroje. Specifikace sémantiky popsané v následujících dvou kapitolách jsou více formální, neboť jsou založeny na logice a matematice, nikoliv na strojích.

2.2 Axiomatická sémantika

Axiomatická sémantika byla vyvinuta v rámci výzkumu zabývajícího se možností dokazování správnosti algoritmů. Důkaz správnosti programu zajistí, že program vykonává algoritmus podle specifikace.

V důkazu programu je každý jeho příkaz předcházen a následován logickým výrazem, který omezuje hodnoty proměnných. Na rozdíl od operační sémantiky tak význam jednotlivých příkazů není definován změnou stavu celého stroje (skutečného či virtuálního), ale pouze několika proměnných. Notace používaná pro popis omezení hodnot proměnných v axiomatické sémantice je predikátový počet. Většinou pro popis omezení hodnot

proměnných dostačují jednoduché logické výrazy, ale někdy jsou jednoduché výrazy nedostatečné.

2.2.1 Tvrzení a nejslabší předpoklad

Logické výrazy (predikáty, tvrzení) předcházející příkazu nazveme předpoklady (preconditions), tvrzení následující bezprostředně za příkazem nazveme důsledky (posconditions). Definice axiomatické sémantiky pro nějaký program znamená, že každému příkazu programu (i programu samotnému) jsou přiřazeny předpoklady a důsledky.

Předpoklady a důsledky bývají zapisovány před a za příkaz programu např. ve složených závorkách.

Příklad zápisu přiřazovacího příkazu s předpokladem a důsledkem:

$$\{ x > 3 \}\ x := x - 3\ \{ x > 0 \}$$

Jestliže po vykonání příkazu m být proměnná x větší než 0 (důsledek), pak předpokladem je, že před jeho vykonáním musí být x větší než 3.

Nejslabší předpoklad (weakest precondition) je nejméně omezující podmínka před příkazem, která ještě zaručí platnost důsledku tohoto příkazu. Např. v předešlém příkladě existuje mnoho podmínek zajišťujících důsledek $x > 0$, např. $x > 10$, $x > 65535$ apod. ovšem nejslabší (nejméně omezující) je právě $x > 3$.

2.2.2 Důkaz správnosti programu

Důkaz správnosti programu může být sestaven, pokud pro každý příkaz může být na základě jeho důsledků určen nejslabší předpoklad. Konstrukce důkazu začíná určením důsledku pro poslední příkaz programu. Z tohoto důsledku je třeba určit nejslabší předpoklad posledního příkazu, který se stává důsledkem předposledního příkazu. Tímto postupem je určen předpoklad prvního příkazu programu, který určuje podmínky za nichž program poskytne požadované výsledky.

Pro některé příkazy je určení nejslabšího předpokladu z důsledku jednoduché a může být specifikováno axiomem. Nicméně ve většině případů ale může být předpoklad spočítán jen pomocí odvozovacího pravidla. Axiom je vždy pravdivý logický výraz. Odvozovací pravidlo je metoda odvození pravdivého tvrzení na základě jiného tvrzení.

Použití axiomatické sémantiky pro nějaký konkrétní programovací jazyk (ať již pro dokazování správnosti programu nebo pro specifikaci sémantiky jazyka) je potřeba pro každý druh příkazu jazyka stanovit buď axiom nebo odvozovací pravidlo.

2.2.3 Přiřazovací příkaz

Jestliže má přiřazovací příkaz podobu $x = E$ a Q je důsledek tohoto příkazu, pak předpoklad P je definován axiomem

$$P = Q_{x\ ?E}$$

který znamená že P je odvozeno s Q se všemi výskyty x nahrazenými E. Například pro přiřazovací příkaz a důsledek

$$a := b / 2 - 1\ \{ a < 10 \}$$

nejslabší předpoklad je spočítán nahrazením $b / 2 - 1$ ve tvrzení $\{ a < 10 \}$:

$$b / 2 - 1 < 10$$

$b < 22$

Tedy nejslabší předpoklad tohoto přiřazovacího příkazu s daným důsledkem je { b < 22 }. Důležité je, že tento axiom pro přiřazovací příkaz platí jen tehdy, pokud tento příkaz nemá žádné vedlejší efekty. To znamená že přiřazovací příkaz nezmění hodnotu jiné proměnné než je uvedena na levé straně přiřazení. U reálných programovacích jazyků, kdy je možné volat v rámci výrazu libovolnou funkci je tato podmínka téměř nesplnitelná.

Pravidlo důsledku (rule of consequence) je odvozovací pravidlo v podobě

$$\frac{S1, S2, \ldots, Sn}{S}$$

říká jestliže S1, S2, …, Sn jsou pravdivé, pak pravdivost S může být odvozena. Pro předpoklad a důsledek lze pravidlo důsledku aplikovat

$$\frac{\{ P \}\ S\ \{ Q \},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{ P' \}\ S\ \{Q' \}}$$

Jestliže tedy { P } S { Q } je pravdivé a P' => P (symbol => značí implikaci) a Q => Q', pak lze odvodit { P' } S {Q' }.

2.2.4 Sekvence

Nejslabší předpoklad sekvence nemůže být stanoven axiomem, protože předpoklady závisí na konkrétních příkazech sekvence. Předpoklady tedy mohou být popsány odvozovacím pravidlem. Nechť S1 a S2 jsou následující příkazy v programu s předpoklady a důsledky:

{ P1 } S1 { P2 }
{ P2 } S2 { P3 }

Odvozovací pravidlo pro takovou sekvenci bude:

$$\frac{\{ P1 \}\ S1\ \{ P2 \},\ \{ P2 \}\ S2\ \{ P3 \}}{\{ P1 \}\ S1; S2\ \{ P3 \}}$$

Pak { P1 } S1; S2 { P3 } popisují axiomatickou sémantiku pro sekvenci S1; S2.

2.2.5 Větvení

Odvozovací pravidlo pro větvení programu má tvar:

$$\frac{\{ B \text{ and } P \}\ S1\ \{ Q \},\ \{ (\text{not } B) \text{ and } P \}\ S2\ \{ Q \}}{\{ P \}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{ Q \}}$$

Toto pravidlo stanoví, že v případě větvení musí být dokázány obě větve. První logický výraz je pro **then** větev, druhý pro **else** větev.

2.2.6 Smyčky

Další základní konstrukcí používanou v programech je smyčka. Zabývejme se jedním druhem smyčky s podmínkou na počátku (while loop). Určení nejslabšího předpokladu pro smyčku je podstatně obtížnější než pro přiřazovací příkaz či příkaz větvení. Postup je založen matematické indukci. Základním krokem je určení indukční hypotézy. V axiomatické sémantice tento krok odpovídá nalezení tvrzení, které je invariantem cyklu. Invariant cyklu je klíčový pro určení nejslabšího předpokladu.

Odvozovací pravidlo pro smyčku s podmínkou na počátku je:

$$\frac{\{ I \text{ and } B \}\ S\ \{ I \}}{\{ I \}\ \text{while } B \text{ do } S \text{ end } \{ I \text{ and } (\text{not } B) \}}$$

kde I je invariant cyklu. Tento invariant musí splňovat řadu podmínek, aby byl použitelný. Nejprve nejslabší předpoklad pro smyčku **while** musí zajistit pravdivost invariantu cyklu. Dále invariant cyklu musí zajistit pravdivost důsledku cyklu po jeho skončení. Dále se invariant nesmí měnit během testování podmínky ukončení cyklu a během vykonávání jednotlivých příkazů těla cyklu (odtud jméno invariant).

Další komplikací smyčky **while** je otázka konečnosti cyklu. Je-li Q důsledek cyklu, pak předpoklad P musí zajistit nejen pravdivou hodnotu Q, ale také ukončení cyklu.

Axiomatická definice smyčky **while** tedy vyžaduje splnění všech následujících podmínek (I je invariant cyklu):

- P => I
- { I } B { I }
- { I and B } S { I }
- (I and (not B)) => Q
- smyčka je konečná

2.3 Významová sémantika (denotational semantics)

Významová sémantika vyvinutá v 70. letech Christopherem Stracheym představuje první matematicky formální popis sémantiky programovacího jazyka (původně byla označovaná matematická sémantika). Formální specifikace sémantiky umožňuje usuzovat na chování programu aniž by bylo nutno spouštět jej na skutečném počítači. Další možností využití specifikace sémantiky je dokazování správnosti (nesprávnosti) programu, důkazy ekvivalence programů apod.

Významová sémantika přiřazuje *význam* nejen celému programu, ale každé *syntaktické frázi* programovacího jazyka (deklarace, příkaz, …). Význam každé fráze je definován pomocí významů jejích podfrází. Specifikace sémantiky tak běží paralelně se syntaktickou strukturou programu.

2.3.1 Základní principy

Smysl každé fráze programu je určen matematickou entitou – *významem*. Ke každé frázi je zapotřebí specifikovat funkci, která jí přiřadí daný význam. Tato funkce je nazývána *sémantická funkce*.

Například čísla jsou v programech zapisována v podobě numerických literálů. Způsob zápisu literálu je definován syntaxí daného jazyka, ale jeho význam je pokaždé stejný. V desítkové soustavě zapíšeme číslo deset literálem *10*. Ovšem můžeme použít jinou soustavu, např. šestnáctkovou *0x0a*. Tento zápis ale odpovídá syntaxi jazyka C, v jazyku Modula-2 by zápis byl v šestnáctkové soustavě *0AH*. Pokaždé se ale jedná o jediné číslo *deset*. Literál (např. *0AH*) je syntaktická entita, hodnota *deset* je sémantická entita.

Příkladem sémantické funkce je např. funkce *ohodnocení* : *literál* → *přirozené číslo*, která přiřazuje význam (hodnotu) literálům.

Vezměme do úvahy syntax velmi jednoduchého kalkulátoru pracujícího s celými čísly:

```
Command    ::=   Expression =
Expression ::=   Literal
               | Expression + Expression
```

```
              | Expression - Expression
              | Expression * Expression
```

Pro definici sémantických funkcí potřebujeme definovat doménu celých čísel:

Integer = { …, -3, –2, -1, 0, 1, 2, 3, … }

a tři pomocné funkce:

sum : *Integer* × *Integer* → *Integer*
diff : *Integer* × *Integer* → *Integer*
mul : *Integer* × *Integer* → *Integer*

Tyto operace jsou obdobou aritmetických operací, ale berou do úvahy limity daného kalkulátoru. Způsobí-li například aritmetická operace přetečení rozsahu kalkulátoru, funkce vrátí ⊥.

Sémantiku kalkulátoru pak formalizují následující funkce:

exec : *Command* → *Integer*
eval : *Expression* → *Integer*
valuate : *Literal* → *Integer*

2.3.2 Lambda kalkul

Významová sémantika používá k zápisu rozšířenou verzi *lambda kalkulu*. Lambda kalkul je velice jednoduchý jazyk, ve své nejjednodušší formě založený pouze na dvou základních konceptech:

- *Abstrakce funkce* v podobě ‘λx.E‘, která označuje funkci s formálním parametrem x (vázanou proměnnou) a výrazem E.
- *Aplikace funkce* (volání funkce) je zapsána ve tvaru ‘E_1 E_2‘. Výsledkem volání E_1 je funkce, které je předán parametr E_2. (Stojí za povšimnutí, že v zápise volání parametr nemusí být nutně v závorkách.)

S těmito dvěma koncepty lze postihnout sémantiku jakéhokoliv výpočtu. Při použití lambda kalkulu nicméně bývá užitečné definovat množinu základních primitiv, např. literály *true* a *false*, 0, 1, 2, … a primitivní funkce *not*, *zero*, *positive*, *succ*, *pred*, *neg*, *add*, *subtract*, *multiply* a *divide*.

Základní lambda kalkul pracuje pouze s funkcemi s jediným argumentem. Např. u funkcí *zero* nebo *neg* to nepřináší problémy, avšak funkce jako *add* typicky pracují se dvěma argumenty. Funkce vyžadující dva argumenty jsou v lambda kalkulu definovány pomocí další pomocné funkce, která je vrácena první funkcí jako výsledek jejího volání s prvním argumentem. Tato pomocná funkce akceptuje druhý argument a vrací konečný výsledek. Tak např. zápis *add m n* je vyhodnocen následovně: funkce *add m* vrátí opět funkci s jediným parametrem, která ke svému argumentu přičítá vždy *m*. Je-li tímto argumentem *n*, výsledkem je součet *m* + *n*. Například volání *add* 2 3 vrátí funkci, která ke svému argumentu přičte číslo 2. Pak (*add* 2) 3 vrátí 5. Tato technika umožní v lambda kalkulu pracovat s funkcemi s libovolným počtem argumentů.

Příklady funkcí v lambda kalkulu:

λ*x*. *x* - identita

λ*n*. *subtract n* 1 - funkce předchůdce

Výrazy typu λ*p*. λ*x*. … jsou vyhodnocovány jako λ*p*. (λ*x*. …)

Formální syntax lambda kalkulu:

```
Expression ::=              SecondaryExpression
                          | λ Identifier . Expression
SecondaryExpression ::=     PrimaryExpression
                          | SecondaryExpression PrimaryExpression
PrimaryExpression ::=       Literal
                          | Identifier
                          | ( Expression )
```

Sémantika lambda kalkulu je obdobně jednoduchá – lambda kalkul pracuje s unárními funkcemi a sadou nadefinovaných primitiv. Se všemi hodnotami lze manipulovat identicky, funkce může být argumentem jiné funkce a stejně tak může být vrácena jako výsledek volání funkce.

Lambda kalkul je netypový jazyk. Například funkce identity ($\lambda x.x$) může být aplikována na jakoukoliv hodnotu – na číslo, na pravdivostní hodnotu, funkci apod.

Lambda kalkul používá statických vazeb. Výskyt identifikátoru bezprostředně za symbolem ‘λ’ značí *vázaný výskyt*, výskyt ve výraze za znakem ‘.’ je *aplikovaný výskyt*. Každý aplikovaný výskyt identifikátoru odpovídá vázanému výskytu v nejbližší funkční abstrakci, která má daný identifikátor jako vázaný. Aplikovaný výskyt identifikátoru x je nazýván *volný* ve výrazu E, pokud neodpovídá žádnému vázanému výskytu x v E.

Výrazy jsou vyhodnocovány podle následujícího pravidla:

$$(\lambda x.E)E' \Rightarrow E[x \leftarrow E']$$

Kde $E[x \leftarrow E']$ je výraz, který obdržíme nahrazením všech volných výskytů x v E za E'.

Příklady vyhodnocování v lambda kalkulu:

$$(\lambda n.\ add\ n\ n)\ \mathbf{3} \Rightarrow add\ 3\ 3 \Rightarrow 6$$

$$(\lambda p.\ \lambda x.\ not(p\ x))\ \boldsymbol{zero} \Rightarrow \lambda x.\ not(zero\ x)$$

$$(\lambda f.\ \lambda x.\ f(f\ x))\ \boldsymbol{succ} \Rightarrow \lambda x.\ succ(succ\ x)$$

$$(\lambda f.\ \lambda x.\ f(f\ x))\ \boldsymbol{succ}\ 7 \Rightarrow (\lambda x.\ succ(succ\ x))\ \mathbf{7} \Rightarrow (succ(succ\ 7)) \Rightarrow 9$$

Pravidla vyhodnocování nejsou přesně předepsána – není určeno kdy vyhodnotit parametr při aplikaci funkce. Jedna možnost je vyhodnotit parametr před jeho dosazením do těla funkce (*včasné vyhodnocení*). Alternativní způsob je nahradit dosud nevyhodnocený parametr v těle funkce (*normální vyhodnocení*).

V řadě případů na způsobu vyhodnocování nezáleží a včasné i normální vyhodnocení vrátí shodný výsledek. Existují ale situace, kdy tomu tak není. Např. funkce

$$(\lambda n.\ 7)\ (\boldsymbol{divide}\ \mathbf{1}\ \mathbf{0})$$

vrátí při normálním vyhodnocení hodnotu 7, kdežto při včasném vyhodnocení selže při pokusu dělit nulou.

Church-Rosserova věta:

- Jestliže výraz E vrátí výslednou hodnotu ν při normálním vyhodnocení, pak při každém jiném způsobu vyhodnocení vrátí buď hodnotu ν nebo vyhodnocení selže.
- Jestliže selže normální vyhodnocení, pak selže i každé jiné vyhodnocení.

Možnost selhání vyhodnocení vynucuje zavedení symbolu $\perp$ jako výsledku výrazu, jehož vyhodnocení selhalo. Pak lze zapsat *divide* 1 0 $\Rightarrow \perp$. Symbolem $\perp$ rovněž označujeme nekonečné vyhodnocení funkce.

Je nutné zdůraznit, že $\perp$ není řádná hodnota a nelze s ní jako s hodnotou zacházet. Funkce můžeme rozdělit na:

- *Striktní*, pokud $f \perp \Rightarrow \perp$.
- *Nestriktní*, pokud výsledkem $f \perp$ nemusí být nutně $\perp$.

Např. funkce *succ* je striktní. Na druhé straně funkce $\lambda n.\ 7$ není striktní, protože $(\lambda n.\ 7)$ $\perp \Rightarrow 7$ při normálním vyhodnocení. Existence nestriktních funkcí činí pořadí vyhodnocování funkcí lambda kalkulu důležitým.

Jako sémantika lambda kalkulu je zvoleno normální vyhodnocení, tedy očekávaný výsledek výrazu je hodnota získaná normálním vyhodnocením. Ve skutečnosti je ale většina funkcí striktních a tak včasné vyhodnocování je u striktních funkcí správné a současně efektivnější.

Lambda kalkul je nejjednodušší možný programovací jazyk. To je důležité pro studium základních konceptů jako vazby a viditelnost, pořadí vyhodnocování, spočitatelnost apod. bez zátěže složité syntaxe a sémantiky skutečných programovacích jazyků. K praktickému využití je ale nutné doplnit základní koncepty známé z jiných jazyků, jako např. podmíněné výpočty, smyčky a rekurzi.

2.3.3 Výrazové prostředky programovacích jazyků

Imperativní programovací jazyky používají proměnné – oblasti v paměti, jejíž obsah se může v průběhu vykonávání programu měnit (např. přiřazovacím příkazem). Tento koncept proměnné měnící hodnotu není v klasické matematice. Ale chování proměnných je možné modelovat sémantickými funkcemi reprezentujícími počítačovou *paměť* coby úložiště hodnot proměnných.

Deklarace zavádějí vazby mezi identifikátory a entitami různého druhu (např. proměnnými či funkcemi). Každá vazba je definována v závislosti na *viditelnosti* identifikátorů různých entit v okamžiku deklarace symbolu. Každá fráze programu se tedy vyskytuje v určitém *prostředí*, což je množina vazeb identifikátorů viditelných v daném okamžiku.

Prostředí je určováno pravidly viditelnosti symbolů konkrétního programovacího jazyka. Na daném programovacím jazyku také záleží množina entit, která může být svázána s identifikátory. Tyto vlastnosti programovacích jazyků lze popsat pomocnými sémantickými funkcemi, které modifikují prostředí v závislosti na deklaraci identifikátorů, vyhledávají entitu podle identifikátoru v daném prostředí apod.

Součástí programovacích jazyků bývají *abstrakce* specifikující nějaký výpočet (procedury a/nebo funkce). Program tyto abstrakce volá, aby provedl výpočet určený procedurou/funkcí.

Na abstrakce funkcí lze nahlížet jako na funkce mapující vstupní argument (z definovaného prostředí) na návratovou hodnotu. Abstrakce procedur lze popsat funkcemi, které vstupní argumenty (z prostředí) mapují do změn paměti. Významnou otázkou zůstávají vedlejší efekty abstrakcí, které sémantický popis programovacího jazyka výrazně komplikují.

Komplikací specifikace pomocných funkcí definujících změny paměti je zavedení složených datových typů (struktur, záznamů). Ačkoliv hodnota složeného typu může být navázána na jediný identifikátor, v principu je možné měnit obsah paměti pro jednotlivé prvky složeného

typu nezávisle. Zavedení složených datových typů vyžaduje změny v definici sémantických funkcí pro základní fráze jazyka.

Příloha A: operační sémantika jazyka OCL

Jazyk OCL svou syntaxí vychází z modulárních jazyků z dílny Niklause Wirtha (Pascal, Modula-2). Je určen především jako bezpečný a jednoduše použitelný (a naučitelný) skriptovací jazyk pro integraci programových komponent. Z těchto důvodů je velmi malý (obsahuje jen několik základních řídicích konstrukcí) a neobsahuje „nebezpečné“ prvky, jako je např. alokace a uvolňování paměti, přístup na adresy (ukazatele) apod. Z těchto důvodů je také velmi vhodný pro demonstraci operační sémantiky.

Syntax jazyka OCL

V zápisu syntaxe jazyka OCL jsou použity tyto typografické konvence:

- nonterminály jsou psány velkými písmeny: PROCEDURE
- terminály (klíčová slova, delimitery, …) jsou tučně: **while ( )**
- hranaté závorky [ ] označují žádný nebo nejvýše jeden výskyt
- kulaté závorky ( ) označují žádný nebo více opakování
- svislá čára | označuje alternativní výskyt
- šipka › odděluje levou a pravou stranu pravidla

```
PROCEDURE      ? procedure ( PARAMETER_LIST ) ;
                 { DECLARATION }
                 begin
                   BLOCK
                 end_procedure ;

PARAMETER_LIST ? [ PARAMETER ] { ; PARAMETER }

PARAMETER      ? [ [ var ] Identifier { , Identifier } : [ array of ] TYPE
]

DECLARATION    ? label LABEL_LIST ;
               | const CONST_LIST ;
               | var VAR_LIST ;
               | static VAR_LIST ;

LABEL_LIST     ? [ Identifier ] { , Identifier }

CONST_LIST     ? [ Identifier = CONST_EXPR ] { ; Identifier = CONST_EXPR }

VAR_LIST       ? [ VARIABLE ] { ; VARIABLE }

VARIABLE       ? Identifier : TYPE [ , CONST_EXPR ]
               | Identifier : array [ CONST_EXPR .. CONST_EXPR ] of TYPE

TYPE           ? boolean
               | integer
               | cardinal
               | real
               | string

BLOCK          ? [ STATEMENT ] { ; STATEMENT }

STATEMENT      ? if EXPR then
                   BLOCK
                 { elsif EXPR then
                   BLOCK }
                 [ else
                   BLOCK ]
                 end
               | loop
                   BLOCK
```

```
                  end
                | while EXPR do
                    BLOCK
                  end
                | repeat
                    BLOCK
                  until EXPR
                | for Variable = EXPR to EXPR [ by CONST_EXPR ] do
                    BLOCK
                  end
                | switch EXPR of
                  { case CONST_EXPR { , CONST_EXPR } :
                    BLOCK }
                  [ else
                    BLOCK ]
                  end
                | goto Identifier
                | return [ EXPR ]
                | Identifier : [ STATEMENT ]
                | Variable := EXPR
                | CALL

STATEMENT       ? exit
                | continue

CALL            ? [ OBJECT_NAME . ] MethodName ( PARAM_LIST )

OBJECT_NAME     ? self | Identifier

PARAM_LIST      ? [ PARAM ] { , PARAM }

PARAM           ? Identifier | EXPR

EXPR            ? SIMPLEEXPR [ RELOP SIMPLEEXPR ]

RELOP           ? = | # | <> | < | > | <= | >=

SIMPLEEXPR      ? TERM { ADDOP TERM }

ADDOP           ? + | - | or | xor | | | ^

TERM            ? FACTOR { MULOP FACTOR }

MULOP           ? * | / | % | and | &

FACTOR          ? ( EXPR )
                | NEGOP FACTOR
                | SIGNOP FACTOR
                | BUILTIN
                | CALL
                | Identifier
                | Identifier [ SIMPLEEXPR ]

NEGOP           ? not | ~

SIGNOP          ? + | -

BUILTIN         ? rand ()
                | sqrt ( SIMPLEEXPR )
                | sin ( SIMPLEEXPR )
                | cos ( SIMPLEEXPR )
                | tan ( SIMPLEEXPR )
                | asin ( SIMPLEEXPR )
                | acos ( SIMPLEEXPR )
                | atan ( SIMPLEEXPR )
                | sinh ( SIMPLEEXPR )
                | cosh ( SIMPLEEXPR )
                | tanh ( SIMPLEEXPR )
                | log ( SIMPLEEXPR )
```

```
| ln ( SIMPLEEXPR )
| exp ( SIMPLEEXPR )
| abs ( SIMPLEEXPR )
| sgn ( SIMPLEEXPR )
| trunc ( SIMPLEEXPR )
| floor ( SIMPLEEXPR )
| ceil ( SIMPLEEXPR )
| frac ( SIMPLEEXPR )
| round ( SIMPLEEXPR )
| pow ( SIMPLEEXPR, SIMPLEEXPR )
| atan2 ( SIMPLEEXPR, SIMPLEEXPR )
| min2 ( SIMPLEEXPR, SIMPLEEXPR )
| max2 ( SIMPLEEXPR, SIMPLEEXPR )
| shr ( SIMPLEEXPR, SIMPLEEXPR )
| shl ( SIMPLEEXPR, SIMPLEEXPR )
| round2 ( SIMPLEEXPR, SIMPLEEXPR )
| val ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| length ( SIMPLEEXPR )
| caps ( SIMPLEEXPR )
| lows ( SIMPLEEXPR )
| pos ( SIMPLEEXPR, SIMPLEEXPR )
| concat ( SIMPLEEXPR, SIMPLEEXPR )
| insert ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| item ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| slice ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| delete ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| str ( SIMPLEEXPR, SIMPLEEXPR )
```

Virtuální stroj OCL

Jazyk OCL je překládán do instrukcí virtuálního stroje, který instrukce interpretuje. Tento stroj není pouze interpreter jednoduchých instrukcí, ale obsahuje některé poměrně vysokoúrovňové abstrakce, které umožňují velmi zjednodušit instrukční soubor a tím výrazně usnadnit překlad.

Příkladem takové abstrakce je matematický výraz. Virtuální stroj neobsahuje elementární instrukce pro matematické operace (sčítání, násobení, …) a matematické výrazy tedy nejsou do takových instrukcí překládány. Namísto toho jsou výrazy přeloženy do stromových struktur, které mohou být jednoduše vyhodnoceny.

V podstatě se tedy jedná o dva virtuální stroje ležící nad sebou. „Nižší“ stroj vyhodnocuje výrazy v podobě stromů. „Vyšší“ stroj interpretuje vlastní instrukce, přičemž s výrazy pracuje jako s abstraktním datovým typem.

Podobně je rozdělen i překladač. Opět existuje „nižší“ překladač schopný analyzovat matematický výraz a „vyšší“ překladač překládá jazyk OCL a k překladu výrazů využívá služeb „nižšího“ překladače. Výhodou tohoto uspořádání (mimo abstrakci vrstev, tedy možností zabývat se jen „nižší“ nebo „vyšší“ vrstvou) je možnost použít překladače výrazů samostatně například při čtení hodnot parametrů (vlastností) komponent, které nejsou definovány v rámci procedury.

Instrukční soubor virtuálního stroje obsahuje tyto instrukce:

```
Jump            // nepodmíněný skok
  Offset        // cíl skoku
JumpTrue        // skok při splnění podmínky
  Offset        // cíl skoku
  Pexpr         // podmínkový výraz
JumpFalse       // skok při nesplnění podmínky
```

```
    Offset          // cíl skoku
    Pexpr           // podmínkový výraz
  JumpTable         // skok podle hodnot v tabulce
    Ncases          // počet položek
    Pcases          // pole položek (obsahuje hodnoty a cíle skoků)
    Offset          // cíl skoku, pokud nevyhovuje žádná položka
    Pexpr           // výraz podle něhož se skáče
  SetVar            // přiřazení proměnné
    Pvar            // proměnná
    Pexpr           // výraz který má být přiřazen
  SetArray          // přiřazení prvku pole
    Pvar            // proměnná typu pole
    Pexpr           // výraz který má být přiřazen
    PindexExpr      // indexový výraz
  Call              // volání metody
    ObjectHandle    // odkaz na objekt
    MethodId        // handle metody
    PParameters     // pole parametrů (výrazů či polí)
    NumParameters   // počet parametrů
  Return            // návrat z procedury
    Pexpr           // výraz návratové hodnoty (může být NIL)
```

Operační sémantika řídicích příkazů OCL definovaná instrukcemi virtuálního stroje

Jako příklad definice sémantiky řídicích příkazů jazyka OCL je uveden překlad řady procedur v jazyce OCL. Tento kód nedává žádný smysl, jen definuje funkčnost použitých řídicích příkazů.

`procedure AssignProc();`
`begin`
`  a = 1;`
`  b[0] = 2;`
`end_procedure;`
	`00000000 SetVar     a, 1`
	`00000010 SetArray   b[ 0 ], 2`
	`00000024 End`

`procedure IfProc();`
`begin`
`  if a = 0 then`
`    Body();`
`  elsif a = 1 then`
`    Body();`
`  else`
`    Body();`
`  end;`
`end_procedure;`
	`00000000 JumpFalse  a = 0, 00000030`
	`0000000C Call       Body()`
	`00000028 Jump       0000007C`
	`00000030 JumpFalse  a = 1, 00000060`
	`0000003C Call       Body()`
	`00000058 Jump       0000007C`
	`00000060 Call       Body()`
	`0000007C End`

`procedure LoopProc();`
`  loop`
`    Body();`
`    continue;`
`    Body();`
`    exit;`
`    Body();`
`  end;`
`end_procedure;`
	`00000000 Call      Body()`
	`0000001C Jump      00000000`
	`00000024 Call      Body()`
	`00000040 Jump      0000006C`
	`00000048 Call      Body()`
	`00000064 Jump      00000000`
	`0000006C End`

`procedure WhileProc();`
`begin`
`  a = 0;`
`  while a < 10 do`
`    a = a + 1;`
`  end;`
`end_procedure;`
	`00000000 SetVar     a, 0`
	`00000010 JumpFalse  a < 10, 00000034`
	`0000001C SetVar     a, a + 1`
	`0000002C Jump       00000010`
	`00000034 End`

`procedure RepeatProc();`
`begin`
`  a = 0;`
`  repeat`
	`00000000 SetVar     a, 0`
	`00000010 SetVar     a, a + 1`
	`00000020 JumpFalse  a = 10, 00000010`
	`0000002C End`

`    a = a + 1;`
`  until a = 10;`
`end_procedure;`	

`procedure ForProc();`
`begin`
`  for a = 1 to 10 by 2 do`
`    Body();`
`  end;`
`end_procedure;`
	`00000000 SetVar     a, 1`
	`00000010 JumpTrue   a > 10, 00000050`
	`0000001C Call       Body()`
	`00000038 SetVar     a, a + 2`
	`00000048 Jump       00000010`
	`00000050 End`

`procedure SwitchProc();`
`begin`
`  switch a of`
`  case 0:`
`    Body();`
`  case 1,2:`
`    Body();`
`  else`
`    Body();`
`  end;`
`end_procedure;`
	`00000000 JumpTable  a`
	`                      0, 0000001C`
	`                      1, 00000040`
	`                      2, 00000040`
	`                      else 00000064`
	`0000001C Call       Body()`
	`00000038 Jump       00000080`
	`00000040 Call       Body()`
	`0000005C Jump       00000080`
	`00000064 Call       Body()`
	`00000080 End`

`procedure GotoProc();`
`label`
`  l1;`
`begin`
`  a = 0;`
`l1:`
`  a = a + 1;`
`  if a < 10 then`
`    goto l1;`
`  end;`
`end_procedure;`
	`00000000 SetVar     a, 0`
	`00000010 SetVar     a, a + 1`
	`00000020 JumpFalse  a < 10, 00000034`
	`0000002C Jump       00000010`
	`00000034 End`

Literatura:

[1] Robert W. Sebesta: Concepts of Programming Languages, 4th ed., Addison Wesley Longman Limited, 1999

[2] David A. Watt: Programming Language Syntax and Semantics, Prentice Hall International (UK) Ltd, 1991

[3] Niklaus Wirth: Compiler Contruction, Addison Wesley Longman Limited, 1996