Principy programovacích jazyků

PB006

Náplň kursu

Přehled programovacích paradigmat, vlastností programovacích jazyků a zkoumání teoretické podstaty jevů, které se v programovacích jazycích vyskytují.

Přednáška

Podzim 2007: čtvrtek 18:00–19:40 D1

Konsultace

Konsultační hodiny (do 21. 12.): úterý 18:30–19:40 v B419

Zkouška

20. 12. 2007 18:00, 16. 1. 2008 15:00, 29. 1. 2008 15:00, 7. 2. 2008 15:00

Aktuální informace

`http://www.fi.muni.cz/~libor/vyuka/PB006/`

Literatura

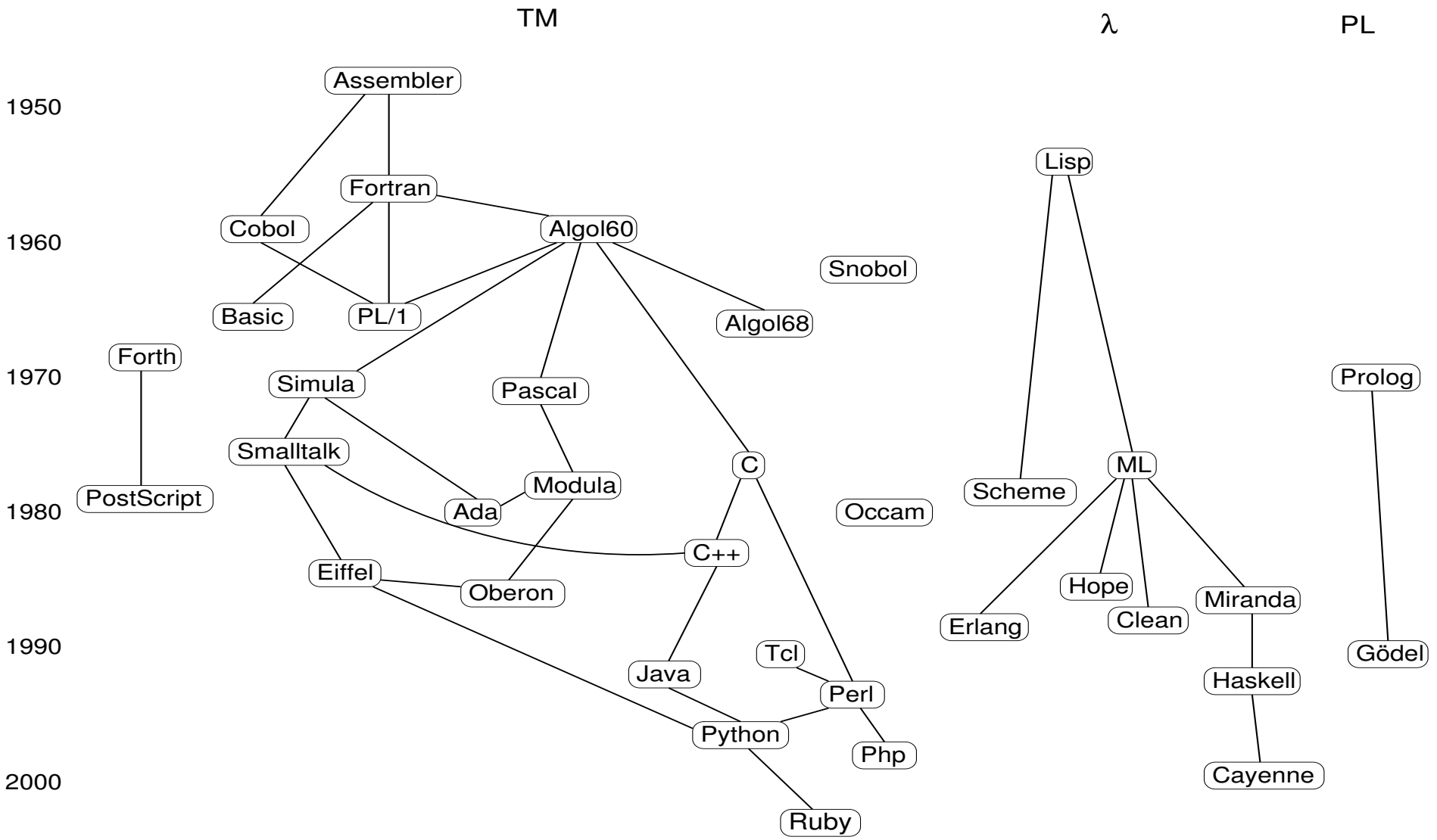
- R. D. Tennent: *Principles of Programming Languages*, Prentice Hall, 1981
- D. A. Watt: *Programming Languages Concepts and Paradigms*, Prentice Hall, 1990
- T. W. Pratt, M. V. Zelkowitz: *Programming Languages – Design and Implementation*, Prentice Hall, 1996

Úvod

Historie programovacích jazyků

Paradigmata

Klasifikace jazyků


TM
λ
PL
1950
1960
1970
1980
1990
2000
Assembler
Fortran
Cobol
Algol60
Basic
PL/1
Algol68
Snobol
Forth
Simula
Pascal
Smalltalk
C
PostScript
Modula
Ada
Occam
C++
Eiffel
Oberon
Tcl
Java
Perl
Python
Php
Ruby
Lisp
Scheme
ML
Hope
Clean
Miranda
Erlang
Haskell
Cayenne
Prolog
Gödel

Klasifikace programovacích jazyků

Paradigma

- imperativní / funkcionální / logické / CLP
- procedurální / objektově orientované
- deterministické / nedeterministické
- sekvenční / paralelní

Struktura programu

- amorfní / blokové / modulární / objektové

Typový systém

- netypované / typované
- odvozování typů: statické / dynamické
- typy:
  * monomorfní / polymorfní
  * paramertický / inklusní / ad hoc polymorfismus

  podtypy, přetížení, druhy, PTS

Vztah k výpočtu

- kompilované / interpretované
- dávkové / interaktivní / knihovní

Oblast nasazení

- číselné výpočty / zpracování textů / syst. programování / simulace / AI / GUI / VR

Doba vzniku

Rozšíření

...

Aspekty programovacího jazyka

Pragmatika

- použitelnost jazyka
- oblast nasazení
- snadnost implementace
- vztah k výpočtu
- úspěšnost jazyka

Syntax

- struktura a forma jazyka
- je popsána formální gramatikou, často ve dvou až třech úrovních (lexikální, bezkontextová, kontext. omezení)
- konkrétní / abstraktní syntax

Sémantika

- význam
- vztah mezi programem a výpočetním modelem
- denotační / operační sémantika
- axiomatická sémantika

Syntax

Konkrétní a abstraktní syntax

Lexikální a frázová syntax

Typová a kontexová omezení

Konkrétní syntax

Popsána formální gramatikou $G = (N, \Sigma, P, S)$,

N, Σ, P konečné, $N \cap \Sigma = \emptyset$, $S \in N$,

$P \subseteq ((N \cup \Sigma)^* \times N \times (N \cup \Sigma)^*) \times (N \cup \Sigma)^*$,

v případě programovacích jazyků tato gramatika bývá bezkontextová, tj.

$P \subseteq N \times (N \cup \Sigma)^+ \cup \{(S, \varepsilon)\}$.

Pro zápis bezkontextových gramatik existují různé formalismy

$$G = \Big(\{\text{Ident}, \text{Písmeno}\}, \{\text{'a'}, \text{'b'}, \text{'\_'}\},$$
$$\{(\text{Ident}, \text{Písmeno}), (\text{Ident}, \text{Ident Písmeno}), (\text{Ident}, \text{Ident \_ Písmeno}),$$
$$(\text{Písmeno}, \text{'a'}), (\text{Písmeno}, \text{'b'})\}, \text{Ident} \Big)$$

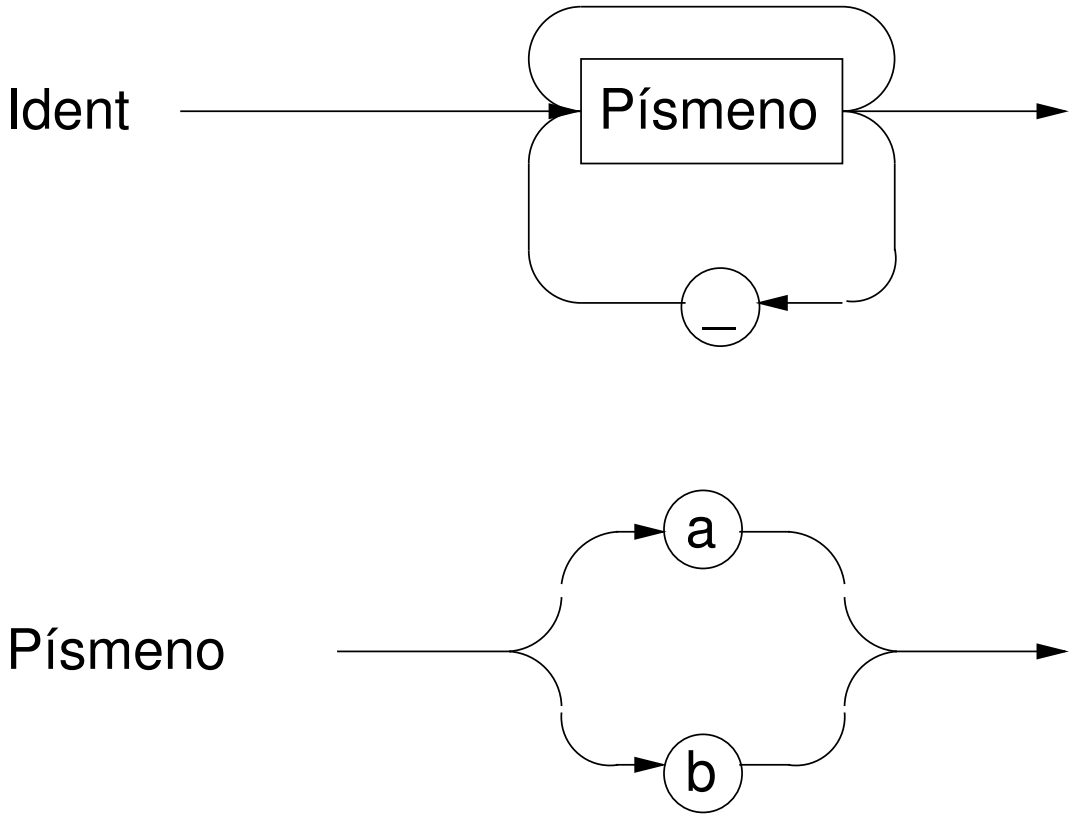
BNF

$$\begin{aligned} \text{Ident} &::= \text{Písmeno} \mid \text{Ident Písmeno} \mid \text{Ident \_ Písmeno} \\ \text{Písmeno} &::= \texttt{a} \mid \texttt{b} \end{aligned}$$

EBNF

$$\begin{aligned} \text{Ident} &::= \text{Písmeno} \left(\text{Písmeno} \mid \text{\_ Písmeno}\right)^* \\ \text{Písmeno} &::= \texttt{a} \mid \texttt{b} \end{aligned}$$

Syntaktický diagram:



Syntax programovacího jazyka bývá obvykle popsána ve více úrovních.

Lexikální syntax – mikrosyntax

Popisuje lexikální strukturu jazyka. Slouží k vymezení lexika – množiny *lexikálních atomů* neboli *lexémů*. Bývá popsána regulární gramatikou nebo jednoduchou bezkontextovou gramatikou. Terminály gramatiky jsou znaky. Vlastní program je pak definován jen jako posloupnost lexikálních atomů.

Mikrosyntax lze přirovnat k morfologii v přirozeném jazyce – tvoří jeho slovník ($\lambda\varepsilon\xi\iota\kappa\acute{o}\nu$).

Kromě toho, že lexikální syntax vymezuje lexikální atomy, popisuje i místo, které je odděluje: tzv. *bílé místo* a *komentáře*. Proto zejména v jazycích se zanořovanými komentáři nestačí pro mikrosyntax regulární gramatika.

Frázová syntax – makrosyntax

Popisuje úplnou strukturu programu. Bývá popsána bezkontextovou gramatikou.

Terminály gramatiky jsou lexikální atomy.

Makrosyntax lze přirovnat k větné skladbě ($\sigma\upsilon\nu\ \tau\acute{\alpha}\xi\iota\varsigma$) přirozeného jazyka – tvoří množinu správně vytvořených vět.

Kontextová omezení

Podmínky vymezující podmnožinu jazyka generovaného gramatikou frázové syntaxe.

Obvykle zaručují, že v programu použité symboly jsou definovány/deklarovány, operátory mají správné arity, program je otypovatelný.

Kontextovým omezením se také říká *statická sémantika*.

Analogií kontextových omezení v přirozeném jazyce jsou pravidla vylučující věty typu „Zelené myšlenky zuřivě spí.“

Makrojazyk

Často bývá programovací jazyk L doplněn druhým jazykem M – tzv. *makrojazykem*, který *předzpracovává* program v jazyce L.

Zdrojové texty programů dávané kompilátoru (nebo interpretu) jsou programy v makrojazyce M. Tyto programy se provádějí (u makrojazyků se říká *expandují*) ještě v době překladu, typicky hned po analýze lexikální syntaxe.

Výsledek *expanze* je program v jazyce L, který jde na vstup analyzátoru frázové syntaxe.

cpp makrojazyk pro C; primitivní, ale lze v něm vyjádřit např. podmíněný překlad nebo vkládání textu z jiných souborů

Scheme v programech lze použít makrojazyk, který má tutéž syntax a stejné funkce jako Scheme

M4 univerzální makrojazyk, použitelný buď samostatně, anebo ve spojení s nějakým programovacím jazykem jako jeho předzpracující makrojazyk

Konkrétní a abstraktní syntax

Konkrétní syntax

- definuje vlastní jazyk
- obsahuje dostatečně mnoho terminálů, které zjednodušují a zjednoznačňují syntaktickou analýzu i čitelnost programu

Abstraktní syntax

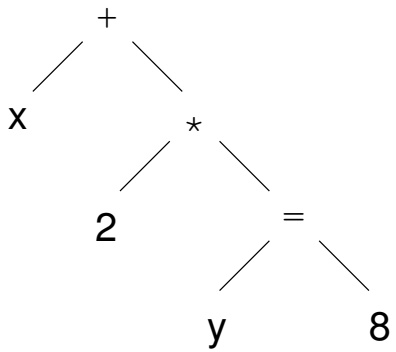
- popisuje strukturu
- obsahuje především neterminály
- terminály slouží pouze jako diskriminátory odvozovacích pravidel
- vhodná pro formální práci s jazykovými termy, např. pro definici denotační sémantiky

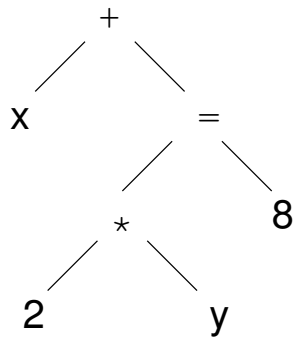
Příklad – abstraktní syntax výrazů

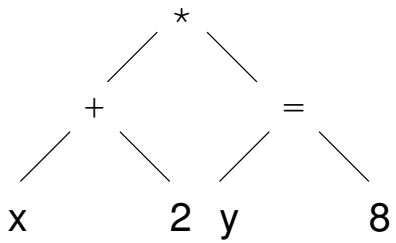
$$E ::= L \mid V \mid E \textbf{ or } E \mid E \textbf{ and } E \mid E = E \mid E < E \mid E + E \mid E * E$$

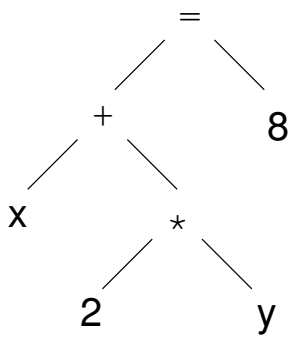
Chápána jako konkrétní syntax je gramatika nejednoznačná, od této nejednoznačnosti však abstrahujeme tím, že místo řetězců pracujeme přímo s termy, jež svou strukturou vyjadřují i odvození příslušného slova

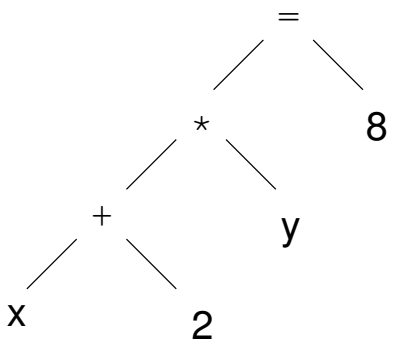
$$(x + (2 * y)) = 8$$





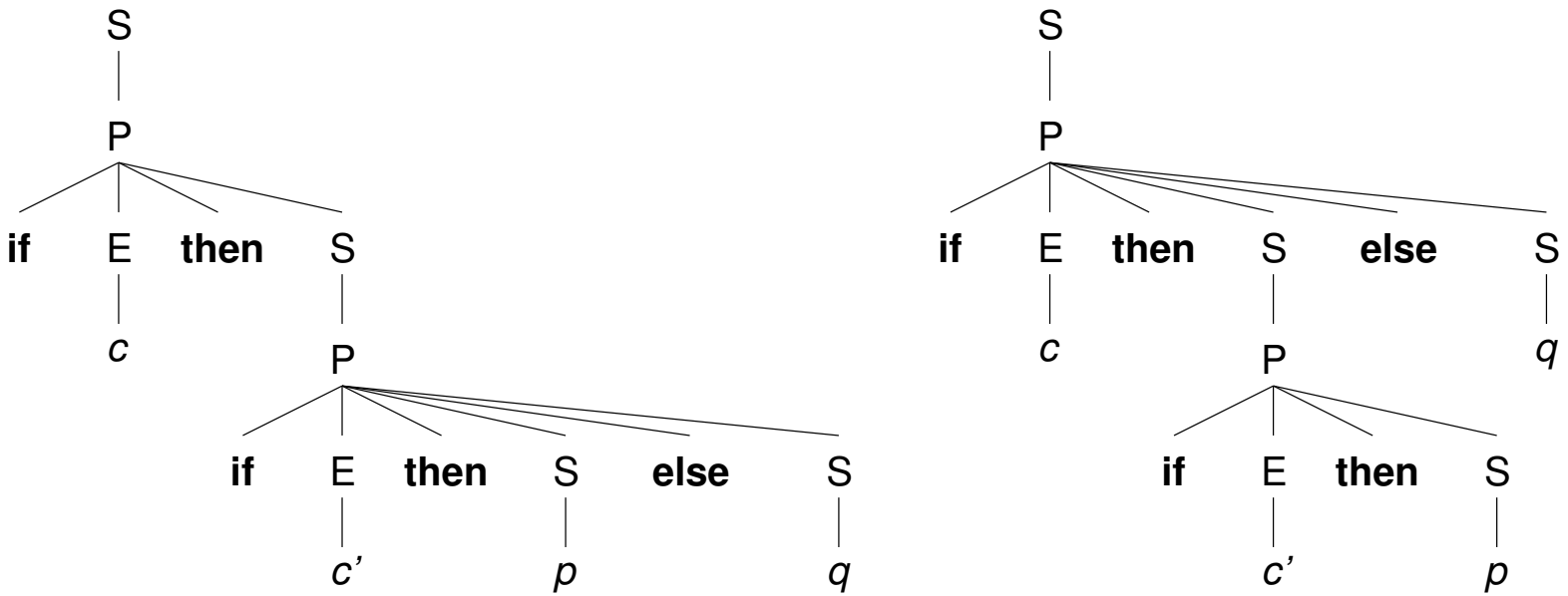






$$
\begin{array}{lcl}
S & ::= & P \mid \ldots \\
P & ::= & \textbf{if } E \textbf{ then } S \mid \textbf{if } E \textbf{ then } S \textbf{ else } S \\
E & ::= & \ldots
\end{array}
$$

if c then if c' then p else q



Abstraktní syntax

Pravidla (bezkontextové) konkrétní syntaxe

$$
\begin{aligned}
A \quad ::= \quad & \alpha_{1,0} B_{1,1} \alpha_{1,1} \ldots \alpha_{1,r_1-1} B_{1,r_1} \alpha_{1,r_1} \\
\mid \quad & \alpha_{2,0} B_{2,1} \alpha_{2,1} \ldots \alpha_{2,r_2-1} B_{2,r_2} \alpha_{2,r_2} \\
\mid \quad & \alpha_{m,0} B_{m,1} \alpha_{m,1} \ldots \alpha_{m,r_m-1} B_{m,r_m} \alpha_{m,r_m}
\end{aligned}
$$

odpovídají disjunktnímu sjednocení

$$\hat{A} = (\hat{B}_{1,1} \times \cdots \times \hat{B}_{1,r_1}) \dot\cup (\hat{B}_{2,1} \times \cdots \times \hat{B}_{2,r_2}) \dot\cup \cdots \dot\cup (\hat{B}_{m,1} \times \cdots \times \hat{B}_{m,r_m})$$

Protože sjednocení je disjunktní, má každý jeho člen jako příznak celé příslušné pravidlo gramatiky (v praxi však většinou zkrácené na nějaký význačný terminál, který toto pravidlo jednoznačně určuje).

Prvky sjednocení $\hat{S}$ odpovídají slovům jazyka, ale i *s jejich derivací*

Proto je abstraktní syntax vždy jednoznačná.

Ale je často zavedena gramatikou, kterou když čteme jako gramatiku konkrétní syntaxe, je tato konkrétní gramatika je nejednoznačná. V bezkontextových jazycích se však tato nejednoznačnost týká pouze struktury vět jazyka a lze ji obejít například vhodným uzávorkováním.

Typy

Základní (primitivní, atomické)

skalární

- logické hodnoty
- znaky (krátké, dlouhé)
- čísla (celá, desetinná, . . .)
- adresy

⋮

Složené

- kartézský součin — *součinové*
- disjunktní sjednocení — *součtové*
- zobrazení (pole, funkce) — *mocninné*

Kartézský součin

$A \times B$

Množina všech uspořádaných dvojic (a, b) takových, že $a : A$, $b : B$.

Funkcím

$$\mathit{fst} : A \times B \to A$$

$$\mathit{fst}\ (x, y) = x$$

$$\mathit{snd} : A \times B \to B$$

$$\mathit{snd}\ (x, y) = y$$

říkáme *projekční funkce* nebo *selektory*.

Je-li $z = (x, y) : A \times B$, pak $x = \mathit{fst}\ z$, $y = \mathit{snd}\ z$.

$A \times B \times C$

Podobně pro kartézský součin tří a více typů.

Agregáty

V typu agregátu jsou definovány projekční funkce, často se zvláštní postfixovou syntaxí.

Je-li

$$\text{Datum} = \{\mathit{den} : \text{Int}, \mathit{mesic} : \text{String}, \mathit{rok} : \text{Int}\}$$

$$d : \text{Datum}$$

$$d = \{\mathit{den} = 5, \mathit{mesic} = \texttt{"říjen"}, \mathit{rok} = 2005\}$$

pak zápis $d.\mathit{den} = 5$ odpovídá $\mathit{den}\,(d) = 5$.

Často syntax jazyka vyžaduje, aby bylo jednoznačně řečeno, *kterého* typu agregátu je daná hodnota:

$$d = \text{Datum}\{\textit{den} = 5, \textit{mesic} = \texttt{"říjen"}, \textit{rok} = 2005\}$$

Disjunktní sjednocení

$A + B$

Sjednocení množiny všech dvojic (L, a) s množinou všech dvojic (R, b), kde $a : A$, $b : B$. L, R jsou příznaky původu prvku a, resp. b.

Funkcím

$$\mathit{inl} : A \to A + B$$

$$\mathit{inl}\ x = (\mathsf{L}, x)$$

$$\mathit{inr} : B \to A + B$$

$$\mathit{inr}\ y = (\mathsf{R}, y)$$

říkáme *inserční funkce* nebo *konstruktory*.

Je-li $x : A$, $y : B$, $u = (\mathsf{L}, x)$, $v = (\mathsf{R}, y)$, pak $u = \mathit{inl}\ x$, $v = \mathit{inr}\ y$.

$A + B + C$

Podobně pro disjunktní sjednocení tří a více typů.

Uniony

V typu unionu jsou definovány inverze konstruktorů, často se zvláštní postfixovou syntaxí. Je-li

$$\text{Cislo} = \{\mathit{cele} : \text{Int} \mid \mathit{desetinne} : \text{Float}\}$$

$$c : \text{Cislo}$$

$$c = \{\mathit{cele} = 42\}$$

pak zápis $c.\mathit{cele} = 42$ odpovídá $\mathit{inl}\ 42 = c$.

Někdy se vyžaduje, aby bylo jednoznačně řečeno, *kterého* typu unionu je daná hodnota:

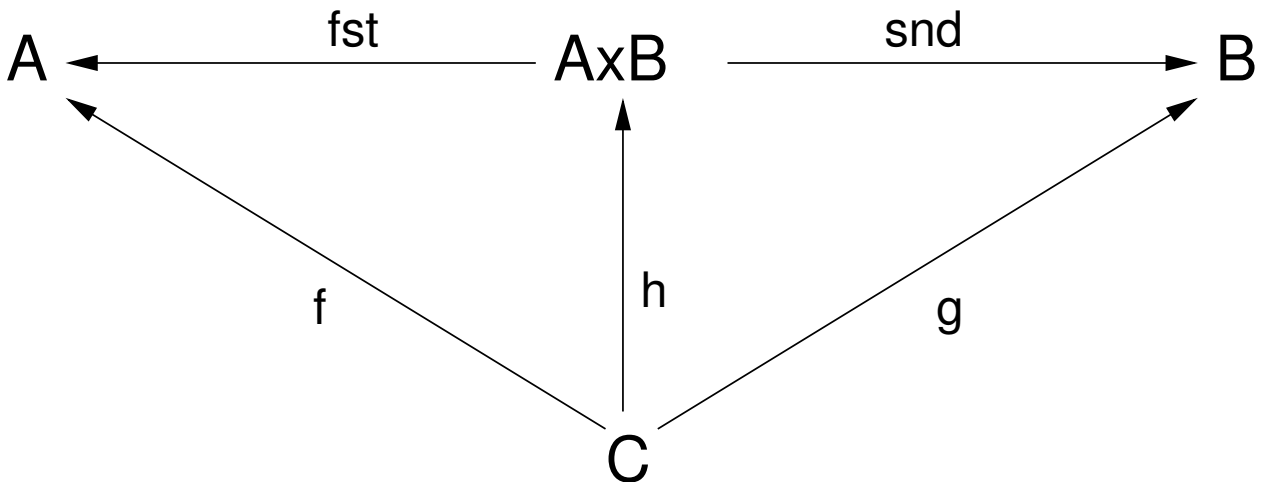
$$c = \text{Cislo}\,\{\mathit{cele} = 42\}$$

Obecné vlastnosti kartézského součinu a disjunktního sjednocení

Nechť A, B, C jsou typy, $f : C \to A$, $g : C \to B$ nějaké dvě funkce.

Pak existuje jediná funkce $h : C \to A \times B$ tak, že

$$\begin{aligned} \mathit{fst} \circ h &= f \\ \mathit{snd} \circ h &= g \end{aligned}$$

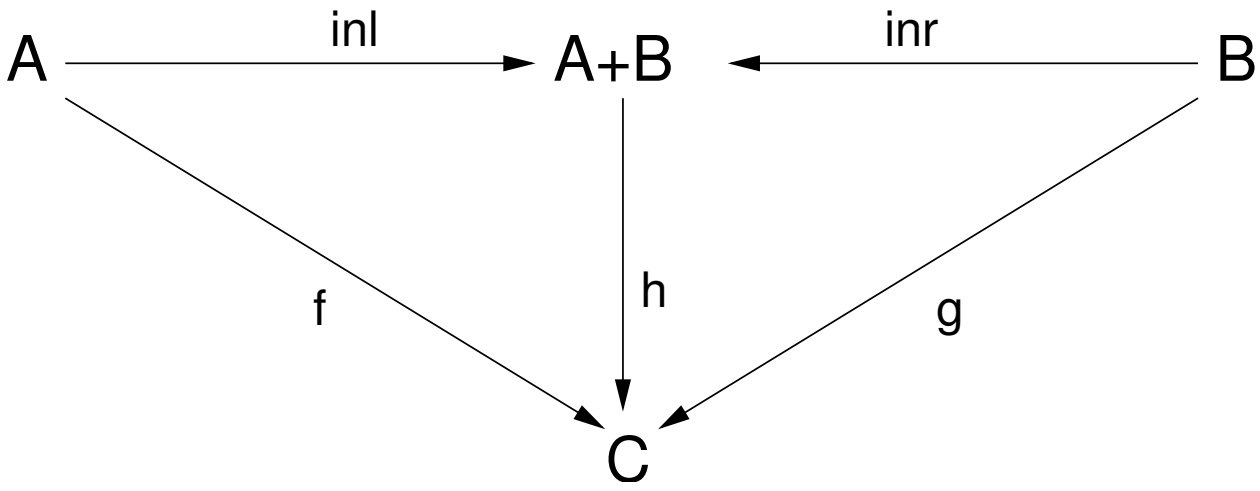


Nechť A, B, C jsou typu, $f : A \to C$, $g : B \to C$ nějaké dvě funkce.

Pak existuje jediná funkce $h : A + B \to C$ tak, že

$$h \circ \mathit{inl} = f$$

$$h \circ \mathit{inr} = g$$



Pole

Zobrazení (konečného) ordinálního typu I do typu T.

(Konečný typ I je ordinální $\Leftrightarrow$ existuje bijekce $ord : I \to \{1, \ldots, n\}$, kde $n = |I|$.)

Zápis: Array $I\ T$

Vícerozměrná pole

$$p = \begin{array}{c} \overbrace{}^{J} \\ I \left\{ \begin{array}{cccc} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \end{array} \right. \end{array} \qquad \text{Array } (I \times J)\ T$$

$$q = I \left\{ \begin{array}{c} \overbrace{}^{J} \\ 11 \quad 12 \quad 13 \quad 14 \\ \overbrace{}^{J} \\ 21 \quad 22 \quad 23 \quad 24 \end{array} \right. \qquad \text{Array } I \ (\text{Array } J\ T)$$

$$p[i, j] \cong q[i][j]$$

Funkce

Zobrazení typu A do typu B.

Zápis: $A \to B$

Funkce více proměnných (větší arity)

$$f = \begin{pmatrix} (1,1) \mapsto 11 & (1,2) \mapsto 12 & (1,3) \mapsto 13 & (1,4) \mapsto 14 \\ (2,1) \mapsto 21 & (2,2) \mapsto 22 & (2,3) \mapsto 23 & (2,4) \mapsto 24 \end{pmatrix} \quad A \times B \to C$$

$$g = \begin{pmatrix} 1 \mapsto & (1 \mapsto 11 & 2 \mapsto 12 & 3 \mapsto 13 & 4 \mapsto 14) \\ 2 \mapsto & (1 \mapsto 21 & 2 \mapsto 22 & 3 \mapsto 23 & 4 \mapsto 24) \end{pmatrix} \quad A \to (B \to C)$$

$$f(x,y) \cong g\ x\ y$$

Potenční množiny

Mocninný typ, speciální případ zobrazení.

Set A	typ všech podmnožin A
$A \to$ Bool	typ všech predikátů nad A

Každé hodnotě a : Set A odpovídá tzv. *charakteristická funkce* $\chi_a : A \to$ Bool taková, že pro každé $x : A$ platí $x \in a \Leftrightarrow \chi_a(x) = \mathit{true}$.

Prázdný typ

Void (disjunktní sjednocení nulového počtu typů)

Neobsahuje žádnou hodnotu.

Jednotkový typ

Unit (kartézský součin nulového počtu typů)

Má jedinou hodnotu – uspořádanou nultici $()$.

Výčtové typy

Zvláštní případ disjunktního sjednocení, kdy všechny inserce jsou typu $\text{Unit} \to T$.

$$\text{Bool} = \{\mathit{false} : \text{Unit} \mid \mathit{true} : \text{Unit}\}$$

$$\text{Tyden} = \{\mathit{po} : \text{Unit} \mid \mathit{út} : \text{Unit} \mid \ldots \mid \mathit{ne} : \text{Unit}\}$$

$$\{\mathit{false} = ()\}, \{\mathit{true} = ()\}, \{\mathit{po} = ()\}, \ldots$$

zkracujeme na

$$\text{Bool} = \mathit{false} \mid \mathit{true}$$

$$\text{Tyden} = \mathit{po} \mid \mathit{út} \mid \mathit{st} \mid \mathit{čt} \mid \mathit{pá} \mid \mathit{so} \mid \mathit{ne}$$

$$\mathit{false}, \mathit{true}, \mathit{po}, \ldots$$

Rekursivní typy

Je-li $F(t)$ zápis typů („typový výraz“) obsahující typovou proměnnou t, pak

$$\mathrm{FIX}\, t.\, F(t)$$

je (množinově) nejmenší typ vyhovující rovnici

$$t = F(t)$$

Definici typu $T = \mathrm{FIX}\, t.\, F(t)$ zapisujeme stručněji

$$T = F(T)$$

a mluvíme o tzv. *rekursivní definici* typu.

Příklad: přirozená čísla

$$\text{Nat} = \text{FIX}\, n.\, \{\mathit{nula} : \text{Unit} \mid \mathit{naslednik} : n\}$$

tj.

$$\text{Nat} = \{\mathit{nula} : \text{Unit} \mid \mathit{naslednik} : \text{Nat}\}$$

je typ s hodnotami

$$\{\mathit{nula} = ()\}$$

$$\{\mathit{naslednik} = \{\mathit{nula} = ()\}\}$$

$$\{\mathit{naslednik} = \{\mathit{naslednik} = \{\mathit{nula} = ()\}\}\}$$

$$\vdots$$

Příklad: seznamy

Je-li T nějaký typ, pak typ $\mathrm{FIX}\ l.\{\mathit{nil} : \mathrm{Unit} \mid \mathit{cons} : T \times l\}$ označujeme T^* a nazýváme typem všech seznamů nad T.

Pro $a_1, a_2, a_3, \ldots$ typu T píšeme

$$
\begin{aligned}
[\,] &= \{\mathit{nil} = ()\} \\
[a_1] &= \{\mathit{cons} = (a_1, \{\mathit{nil} = ()\})\} \\
[a_1, a_2] &= \{\mathit{cons} = (a_1, \{\mathit{cons} = (a_2, \{\mathit{nil} = ()\})\})\} \\
&\vdots
\end{aligned}
$$

Monomorfismus a polymorfismus

Monomorfní typy

jsou buď základní typy, anebo jsou (pomocí typových konstruktorů) složené z monomorfních typů.

Polymorfní typy

zastupují celou množinu monomorfních typů.

Parametrický polymorfismus – v typových výrazech se vyskytují typové proměnné, za něž lze dosadit libovolný typ.

Inklusní (podtypový) polymorfismus – v typovém systému se zavede mezi typy relace $\leqslant$: a každá hodnota má kromě svého nejmenšího typu i všechny jeho nadtypy.

Parametrický polymorfismus

Typ je parametricky polymorfní, zastupuje-li celou množinu monomorfních typů. Tato množina je generována náhradou typových proměnných všemi monomorfními typy (v typovém výrazu reprezentujícím polymorfní typ). Parametricky polymorfní typ je vyjádřen typovým výrazem se základními typy, typovými konstruktory a universálně kvantifikovanými typovými proměnnými.

Příklady:

$$\forall b.\ \text{Array}\ (0..9)\ b$$

$$\forall a.\ a \rightarrow \text{Bool}$$

$$\forall a \forall b \forall c \ldots \forall o.\ \{asel : a,\ bsel : b,\ csel : c,\ \ldots,\ osel : o\}$$

Typ T' vzniklý z parametricky polymorfního typu T náhradou za nějakou jeho typovou proměnnou se nazývá *odvozený* z T. Říkáme pak, že T je *obecnější* než T'.

Hlavní typ hodnoty v je její nejobecnější typ, z něhož lze odvodit pouze typy, které jsou v souladu s definicí hodnoty v.

Příklady:

$$\mathit{fst}\ (x, y) = x \qquad \mathit{fst} : \forall a \forall b.\ a \times b \to a$$

$$\mathit{id}\ (x) = x \qquad \mathit{id} : \forall a.\ a \to a$$

Množina hodnot typu T (tj. množina těch hodnot, jichž je T hlavním typem) je rovna průniku množin hodnot všech monomorfních typů odvozených z T.

$a \to a$	$\{\mathit{id}\}$
$a \to b \to a$	$\{\mathit{const}\}$
$(a, b) \to b$	$\{\mathit{snd}\}$
$[a]$	$\{[\,]\}$

Polymorfní typové systémy
- impredikativní (Girard-Reynolds)
- predikativní (Hindley-Milner)

Otypování výrazu je nalezení jeho (hlavního) typu. V monomorfních typových systémech je triviální.

Otypování v parametricky polymorfních typových systémech je založeno na unifikaci typových výrazů. Pro složitější typové systémy je obecně nerozhodnutelné.

Typové systémy s podtypy

Na množině typů je zavedeno uspořádání $\leqslant:$.

Zápis $A \leqslant: B$ čteme „A je *podtypem* typu B“.

Je-li $A \leqslant: B$, pak v kontextu, ve kterém se očekává hodnota typu B, můžeme bezpečně použít hodnotu podtypu A.

Pravidlo subsumpce

Nechť A, B jsou typy takové, že $A \leqslant: B$, a nechť a je hodnota typu A.

Potom také a je typu B.

Pravidlo subsumpce se zapisuje ve tvaru

$$\frac{a : A \qquad A \leqslant: B}{a : B}$$

Kovariance a kontravariance

Typový konstruktor S je *kovariantní*, když

$$\frac{A \leqslant: A'}{S\ A \leqslant: S\ A'}$$

Typový konstruktor T je *kontravariantní*, když

$$\frac{A \leqslant: A'}{T\ A' \leqslant: T\ A}$$

Jinak je typový konstruktor (v daném typovém parametru) *invariantní*.

Příklad: Typový konstruktor Set je kovariantní:

$$\sigma \leqslant: \sigma' \Rightarrow \text{Set}\ \sigma \leqslant: \text{Set}\ \sigma'$$

U typových konstruktorů větší arity (binárních, ternárních…) se kovariance a kontravariance zavádí zvlášť pro každý typový parametr.

n-ární typový konstruktor S je ve svém i-tém typovém parametru ($1 \leq i \leq n$) *kovariantní*, když

$$\frac{A_i \leqslant: A'}{S\ A_1\ A_2\ \ldots\ A_n \leqslant: S\ A_1\ A_2\ \ldots\ A_{i-1}\ A'\ A_{i+1}\ \ldots\ A_n}$$

n-ární typový konstruktor T je ve svém i-tém typovém parametru ($1 \leq i \leq n$) *kontravariantní*, když

$$\frac{A_i \leqslant: A'}{T\ A_1\ A_2\ \ldots\ A_{i-1}\ A'\ A_{i+1}\ \ldots\ A_n \leqslant: T\ A_1\ A_2\ \ldots\ A_n}$$

Příklad: Typový konstruktor $\rightarrow$ je v prvním parametru kontravariantní a ve druhém parametru kovariantní:

$$\sigma \leqslant: \sigma' \quad \Rightarrow \quad \sigma' \rightarrow \tau \leqslant: \sigma \rightarrow \tau$$
$$\tau \leqslant: \tau' \quad \Rightarrow \quad \sigma \rightarrow \tau \leqslant: \sigma \rightarrow \tau'$$

Z kovariance druhého typového parametru a pravidla subsumpce vyplývá, že každou funkci typu $\sigma \rightarrow \tau$ lze považovat i za funkci nadtypu $\sigma \rightarrow \tau'$. Tedy volání $f(a)$, které vrací hodnotu typu τ, se může objevit v kontextu, kde se očekává hodnota nadtypu τ'.

Z kontravariance prvního typového parametru a pravidla subsumpce vyplývá, že každou funkci typu $\sigma' \rightarrow \tau$ lze považovat i za funkci nadtypu $\sigma \rightarrow \tau$. Tedy očekává-li funkce f argument typu σ', lze ji aplikovat na argument jeho podtypu σ.

Využití: zejména v typovaných systémech pro objektově orientované jazyky.

Nevýhoda: nerozhodnutelnost otypování $\Rightarrow$ nutné dynamické typové kontroly.

Inklusní polymorfismus

Též *podtypový polymorfismus*. Je důsledkem subsumpčního principu. Totiž je-li $a : A$, pak typem hodnoty a je i každý nadtyp typu A.

Př.:

$$\text{Bod} = \{x : \text{Real};\ y : \text{Real}\}$$

$$\text{BarevnyBod} = \{x : \text{Real};\ y : \text{Real}, c : \text{Barva}\}$$

$$\text{BarevnyBod} \leqslant: \text{Bod}$$

$$\textit{presun} : \text{Bod} \to \text{Bod}$$

$$\textit{presun}\,\{x = p;\ y = q\} = \{x = p + 1;\ y = q + 1\}$$

$$\textit{zmodrej} : \text{Bod} \to \text{BarevnyBod}$$

$$\textit{zmodrej}\,\{x = p;\ y = q\} = \{x = p;\ y = q;\ c = \textit{modra}\}$$

Potom také

$$\{x = 2;\ y = 3;\ c = \textit{zelena}\} : \text{Bod}$$

$$\textit{presun} : \text{BarevnyBod} \to \text{Bod}$$

$$\textit{zmodrej} : \text{BarevnyBod} \to \text{BarevnyBod}$$

$$\textit{zmodrej} : \text{Bod} \to \text{Bod}$$

$$\textit{zmodrej} : \text{BarevnyBod} \to \text{Bod}$$

Přetížení

Symbol je přetížený, označuje-li více hodnot různých typů. Obvykle je každá z těchto hodnot definována různě.

Jednodušší jazyky mají přetížené jen zabudované operátory, jazyky s bohatšími typovými systémy řeší přetížení systematicky (např. pomocí typových tříd) a umožňují definovat nová jména přetížená více hodnotami.

Při překladu nebo interpretaci je nutno přetížené symboly takzvaně *vyřešit* – obdařit sémantikou *jediné* z hodnot, jež ho přetěžují. V konkrétním výrazu se pozná, která z těchto hodnot to bude (například z počtu argumentů přetížené funkce a z jejich typů).

U funkcí a procedur se též rozlišuje *kontextově nezávislé* a *kontextově závislé* přetížení.

Kontextově nezávisle přetížené funkce a procedury lze vyřešit jen z typů argumentů.

U kontextově závisle přetížených funkcí (procedur) typy argumentů nestačí a je nutno vzít do úvahy širší kontext.

Nechť symbol g je přetížen dvěma různými typy $\sigma_1 \to \tau_1,\ \sigma_2 \to \tau_2$.

- V kontextově nezávislém přetížení (Pascal, Haskell, ML) musí být $\sigma_1 \neq \sigma_2$.
- V kontextově závislém přetížení (Ada) je $\sigma_1 = \sigma_2$, ale musí být $\tau_1 \neq \tau_2$. Nelze je vždy jednoznačně vyřešit. Jazyk musí nejednoznačné výrazy zakazovat.

Příklad přetížení

Nechť operátor $(/)$ je přetížen třemi typy: $(/) : \text{Int} \times \text{Int} \to \text{Int}$

$(/) : \text{Float} \times \text{Float} \to \text{Float}$

$(/) : \text{Int} \times \text{Int} \to \text{Float}$

Přetížení je kontextově závislé. Nechť $x : \text{Float}$, $n : \text{Int}$ a nechť $(=)$ je (rovněž přetížený) operátor rovnosti vyžadující, aby obě strany rovnosti byly stejného typu.

$$
\begin{array}{lcl}
x = 7.0/2.0 & \Rightarrow & x = 7.0/2.0 \\
x = 7/2 & \Rightarrow & x = 7/2 \\
n = 7/2 & \Rightarrow & n = 7/2 \\
n = (7/2)/(5/2) & \Rightarrow & n = (7/2)/(5/2) \\
x = (7/2)/(5/2) & \Rightarrow & x = (7/2)/(5/2) \\
 & \text{nebo} & x = (7/2)/(5/2)
\end{array}
$$

Poslední výraz je nejednoznačný, přetížení nelze vyřešit.

Koerce

Koerce je implicitní zobrazení hodnot jednoho typu do hodnot jiného typu.

Int $\rightarrow$ Float, Float $\rightarrow$ Double, Float $\rightarrow$ Complex (widening)

$a \rightarrow (a \mid b)$ (unioning)

$a \rightarrow ()$ (forgetting)

$a \rightarrow$ Array $i\ a$ (rowing, autoboxing)

Ref $a \rightarrow a$ (dereferencing)

V moderních jazycích se od koerce upouští – interferuje s přetížením a polymorfismem.

Hodnotově závislé typy

Bohatší typové systémy mohou pracovat s typy závislými na hodnotách. Typové konstruktory mohou být parametrizovány nejen typy, ale i hodnotami.

Příklad – skalární součin

V jazycích bez závislých typů lze test na rovnost délek násobených vektorů buď neprovádět vůbec

$$\begin{aligned} \mathit{dp}\ []\ [] &= 0 \\ \mathit{dp}\ s\ t &= \mathit{head}\ s * \mathit{head}\ t + \mathit{dp}\ (\mathit{tail}\ s)\ (\mathit{tail}\ t) \end{aligned}$$

anebo ho posunout do doby běhu.

V jazycích se závislými typy lze korektnost zaručit staticky tím, že informaci o délce vektoru zahrneme do jeho typu:

$$\begin{aligned} \mathit{vhead} &: \forall n : \mathsf{Nat}.\ \mathsf{Vec}\ (n+1) \to \mathsf{Float} \\ \mathit{vtail} &: \forall n : \mathsf{Nat}.\ \mathsf{Vec}\ (n+1) \to \mathsf{Vec}\ n \end{aligned}$$

$$\begin{aligned} \mathit{dp} &: \forall n : \mathsf{Nat}.\ \mathsf{Vec}\ n \to \mathsf{Vec}\ n \to \mathsf{Float} \\ \mathit{dp}\ []\ [] &= 0.0 \\ \mathit{dp}\ s\ t &= \mathit{vhead}\ s * \mathit{vhead}\ t + \mathit{dp}\ (\mathit{vtail}\ s)\ (\mathit{vtail}\ t) \end{aligned}$$

Příklad – typ funkce *printf*

$$\text{Printf} \;:\; \text{String} \to *$$

$$\begin{aligned}
\text{Printf}\,(\texttt{"\%d"} \mathbin{++} t) &= \text{Int} \to \text{Printf}\; t \\
\text{Printf}\,(\texttt{"\%s"} \mathbin{++} t) &= \text{String} \to \text{Printf}\; t \\
\text{Printf}\; t &= \text{Printf}\; t' \;, && \text{nezačíná-li } t \text{ znakem \% a } t' \text{ je zbytek} \\
\text{Printf}\; \texttt{""} &= \text{String}
\end{aligned}$$

$$\mathit{printf} \;:\; (t : \text{String}) \to \text{Printf}\; t$$

$$\mathit{printf}\; t \;=\; \mathit{pr}\; t\; \texttt{""}, \qquad \text{kde} \quad
\begin{aligned}
\mathit{pr}\; \texttt{""}\; v &= v \\
\mathit{pr}\,(\texttt{"\%d"} \mathbin{++} t)\; v &= \lambda(i{:}\text{Int}) \mapsto \mathit{pr}\; t\; (v \mathbin{++} \mathit{show}\; i) \\
\mathit{pr}\,(\texttt{"\%s"} \mathbin{++} t)\; v &= \lambda(s{:}\text{String}) \mapsto \mathit{pr}\; t\; (v \mathbin{++} s) \\
\mathit{pr}\,(u \mathbin{++} t)\; v &= \mathit{pr}\; t\; (v \mathbin{++} u)\;, \quad \text{jinak}
\end{aligned}$$

$$\mathit{printf}\; \texttt{"ab"} \;:\; \text{Printf}\; \texttt{"ab"} \;=\; \text{String}$$

$$\mathit{printf}\; \texttt{"\%s=\%d"} \;:\; \text{Printf}\; \texttt{"\%s=\%d"} \;=\; \text{String} \to \text{Int} \to \text{String}$$

Výhody hodnotově závislých typů

Mnoho vlastností lze závislými typy vyjádřit jemněji a přesněji.

Pomocí typů lze vyjádřit sémantiku. Potom otypování = důkaz korektnosti.

Nevýhody

Otypování je *nerozhodnutelné*.

Jazyky

Vesměs akademické jazyky:

Agda, Cayenne, Dependent ML, Russell, Epigram ...

Pojetí typů

- množiny hodnot
- heterogenní algebry (množiny hodnot spolu s operacemi na nich) — algebraické typy
- variety heterogenních algeber (množiny s operacemi a axiomy) — ADT

Příklad: Typ zásobník nad celými čísly

1. Zásobníky jsou pevným způsobem reprezentovány, například pomocí pole a indexu vrcholu zásobníku. Typ Zásobník má význam *množiny* všech takových dat (polí konkrétních hodnot + indexů).

   Nad zásobníky lze definovat externí operace závislé na reprezentaci.

2. Typ Zásobník má význam množiny všech *heterogenních univerzálních algeber* se signaturou

   Sorty: Zásobník, Číslo, Boolean

   Operace:

$$\textit{prázdný} : \text{Zásobník}$$

$$\textit{jeprázdný} : \text{Zásobník} \rightarrow \text{Boolean}$$

$$\textit{push} : \text{Zásobník} \times \text{Číslo} \rightarrow \text{Zásobník}$$

$$\textit{pop} : \text{Zásobník} \rightarrow \text{Zásobník}$$

$$\textit{top} : \text{Zásobník} \rightarrow \text{Číslo}$$

3. Typ Zásobník má význam *variety* všech heterogenních univerzálních algeber, která je určena teorií

   Sorty

   Operace

   Axiomy:

$$\mathit{jeprázdný}(\mathit{prázdný}) = \mathit{True}$$

$$\mathit{jeprázdný}(\mathit{push}(z, n)) = \mathit{False}$$

$$\mathit{pop}(\mathit{push}(z, n)) = z$$

$$\mathit{top}(\mathit{push}(z, n)) = n$$

Sémantika

Axiomatická sémantika

Operační sémantika

Denotační sémantika

Sémantika ($\sigma\eta\mu\alpha\nu\tau\iota\kappa\acute{o}\varsigma$) přiřazuje významy programům nebo jejich částem.

Sémantiku programovacího jazyka lze zadat

- Neformálně – popisem v přirozeném jazyce. To může být nepřesné, často je popis neúplný, nejednoznačný.
- Prohlášením jedné implementace kompilátoru za standard. Sice jednoznačné, ale případné chyby v kompilátoru se tak stávají součástí definice jazyka.
- Formálně, pomocí matematického zápisu.

Formální sémantika

Denotační sémantika je definovaná na programech a jejich komponentách explicitně (jako množina funkcí přiřazujících částem programů význam).

Operační sémantika se zavádí množinou pravidel popisujících výpočet abstraktního počítače. Výsledek tohoto výpočtu je významem programu.

Axiomatická sémantika je dána jako množina tvrzení (tzv. teorie) o nějakém systému, v němž probíhá výpočet. Používá se zejména u imperativních jazyků, protože tvrzení se vyjadřují o stavech.

Denotační sémantika – příklad

MVar	množina programových (přepisovatelných) proměnných
V	množina hodnot
$S = \mathit{MVar} \longrightarrow V$	množina stavů
$[\![\_]\!]\_ : P \times S \longrightarrow S$	stavový transformátor
$\mathcal{M}[\![\_]\!]\_ : P \times S \longrightarrow V$	významová funkce

$$[\![x\,{+}{+}]\!]\sigma = \sigma', \quad \text{kde} \quad \sigma'(x) = \sigma(x) + 1$$

$$\forall y, y \neq x.\ \sigma'(y) = \sigma(y)$$

$$\mathcal{M}[\![x\,{+}{+}]\!]\sigma = \sigma(x)$$

Operační sémantika – příklad

Relace „krok výpočtu“ $\rightsquigarrow \subseteq (P \times S) \times (P \times S)$ je zadána množinou pravidel.

$$\frac{\langle e, \sigma \rangle \rightsquigarrow \langle \mathit{tt}, \sigma' \rangle}{\langle \textbf{if } e \textbf{ then } p \textbf{ else } q,\ \sigma \rangle \rightsquigarrow \langle p, \sigma' \rangle}$$

$$\frac{\langle e, \sigma \rangle \rightsquigarrow \langle \mathit{ff}, \sigma' \rangle}{\langle \textbf{if } e \textbf{ then } p \textbf{ else } q,\ \sigma \rangle \rightsquigarrow \langle q, \sigma' \rangle}$$

$$\frac{\langle e, \sigma \rangle \rightsquigarrow \langle e', \sigma' \rangle}{\langle \textbf{if } e \textbf{ then } p \textbf{ else } q,\ \sigma \rangle \rightsquigarrow \langle \textbf{if } e' \textbf{ then } p \textbf{ else } q, \sigma' \rangle}$$

Axiomatická sémantika – příklad

Sekvent v Hoarově logice je trojice tvaru $\{A\}\ p\ \{B\}$, kde A je tzv. *vstupní podmínka*, B je *výstupní podmínka* a p je příkaz. Množina *odvoditelných sekventů* je zadána odvozovacími pravidly a axiomy.

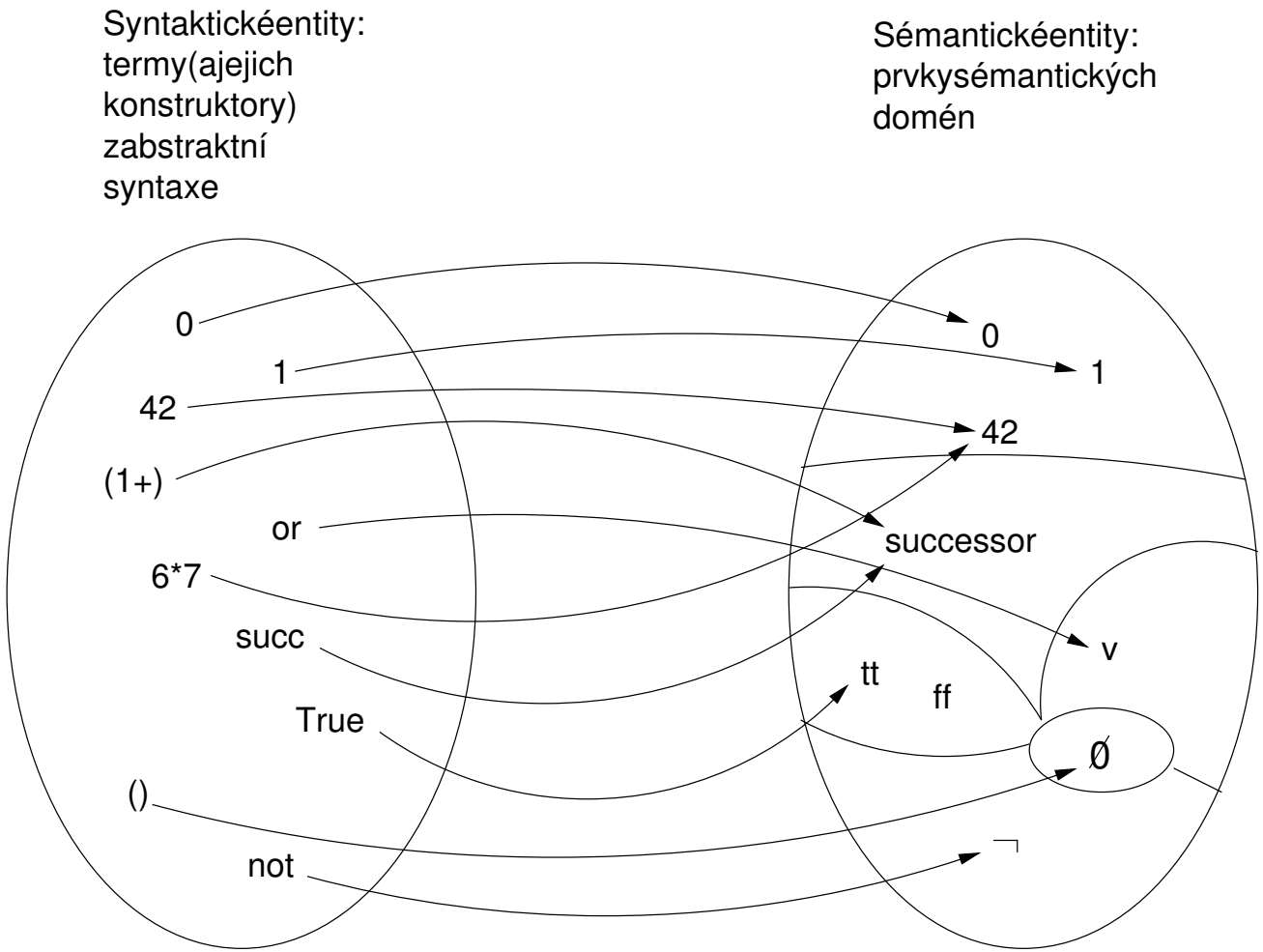
$$\frac{}{\{[e/\xi]A\}\ x := e\ \{[!x/\xi]A\}}$$

$$\frac{\{A\}\ p\ \{B\},\quad B \supset C,\quad \{C\}\ q\ \{D\}}{\{A\}\ \textbf{begin}\ p\,;q\ \textbf{end}\ \{D\}}$$

$$\frac{\{A \wedge e\}\ p\ \{A\}}{\{A\}\ \textbf{while}\ e\ \textbf{do}\ p\ \{A \wedge \neg e\}}$$

Denotační sémantika

Přiřazení prvků sémantických domén programovým konstrukcím



Nedeterministická sémantika

Příkazy určují nedeterministické stavové transformátory:

$$s : S \to \mathcal{P}(S)$$

Sekvence

$$[\![\textbf{begin end}]\!]\sigma = \{\sigma\}$$

$$[\![\textbf{begin } p_1; \ldots; p_k \textbf{ end}]\!]\sigma = \bigcup \{[\![p_k]\!]\tau | \tau \in [\![\textbf{begin } p_1; \ldots; p_{k-1} \textbf{ end}]\!]\sigma\}$$

pro $k \geq 1$

Nedeterministický výběr

$$[\![\textbf{begin } p_1 \mid \ldots \mid p_k \textbf{ end}]\!]\sigma = \bigcup_{i=1}^{k} [\![p_i]\!]\sigma$$

Paralelní kompozice

$$[\![\textbf{begin } p_1 \| \ldots \| p_k \textbf{ end}]\!]\sigma = \bigcup_{\iota \text{ permutace}} [\![\textbf{begin } p_{\iota(1)}; \ldots; p_{\iota(k)} \textbf{ end}]\!]\sigma$$

pro atomické $p_1, \ldots, p_k$

Deterministická sémantika

Významem výrazu e je jeho hodnota (prvek sémantické domény).

V imperativních jazycích závisí sémantika výrazu na *stavu* $\sigma \in S$, takže

$$\mathcal{M}[\![e]\!] : S \longrightarrow \mathit{Val}$$

Význam výrazu e ve stavu σ je hodnota $\mathcal{M}[\![e]\!]\sigma \in \mathit{Val}$.

V jazycích bez vedlejších efektů, ale s (čistými) proměnnými závisí sémantika výrazu na tzv. *hodnotovém kontextu* $\varepsilon \in \mathit{Env}$. Potom

$$\mathcal{M}[\![e]\!] : \mathit{Env} \longrightarrow \mathit{Val}$$

Význam výrazu e v hodnotovém kontextu ε je hodnota $\mathcal{M}[\![e]\!]\varepsilon \in \mathit{Val}$.

Sémantika výrazů se definuje pomocí sémantiky podvýrazů.

Příklad:

$$
\begin{aligned}
\mathcal{M}[\![\mathtt{0}]\!]\sigma &= 0 \\
\mathcal{M}[\![\mathtt{1}]\!]\sigma &= 1 \\
&\vdots \\
\mathcal{M}[\![e_1 \mathtt{+} e_2]\!]\sigma &= \mathcal{M}[\![e_1]\!]\sigma + \mathcal{M}[\![e_2]\!]\sigma \\
\mathcal{M}[\![e_1 \mathtt{*} e_2]\!]\sigma &= \mathcal{M}[\![e_1]\!]\sigma \cdot \mathcal{M}[\![e_2]\!]\sigma \\
\mathcal{M}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\sigma &= \begin{cases} \mathcal{M}[\![e_2]\!]\sigma, \text{ pokud } \mathcal{M}[\![e_1]\!]\sigma = \textit{true} \\ \mathcal{M}[\![e_3]\!]\sigma, \text{ pokud } \mathcal{M}[\![e_1]\!]\sigma = \textit{false} \\ \bot \text{ jinak} \end{cases}
\end{aligned}
$$

Výrazy

Literály označují hodnoty a obvykle jsou tvořeny jediným lexikálním atomem se zvláštní lexikální syntaxí, například *42*, *1.602e−19*, *0xBE*, ’*’, ’\n’.

Jména

- (pojmenované) konstanty: *pi*, *sin*
- parametry funkcí
- (přepisovatelné) proměnné

Výrazy popisující hodnoty **složených typů**

$(1, \text{'a'})$, $\{\mathit{jmeno} = \texttt{"Bob"}; \mathit{adresa} = \texttt{"klobouk"}\}$

Podmíněné výrazy

if . . . **then** . . . **else**, **case** . . . **of** . . .

Aplikace funkce na argumenty, tzv. „volání funkce“.

Abstrakce („uzávěry“).

Příkazy v imperativních jazycích lze považovat za zvláštní druh výrazu (výraz typu Command).

Funkcionální a procedurální abstrakce

V mnoha jazycích lze funkcionální/procedurální abstrakci použít jen v definicích funkcí/procedur, ale např. v Algolu68 je lze použít i ve výrazech/příkazech a ve funkcionálních jazycích se funkcionální abstrakce (lambda abstrakce) užívají běžně.

Tzv. procedury lze považovat za funkce typu $t_1 \times \cdots \times t_n \to$ Command.

Př.:

$$\begin{aligned} \textit{repeat} &: \text{Procedure}\big(\text{Int}, \text{Command}\big); \\ \textit{mezery} &: \text{Procedure}\big(\text{Int}\big); \end{aligned}$$

$$\begin{aligned} \textit{repeat} &= \textbf{proc}\big(n : \text{Int}, p : \text{Command}\big)\big\{\textbf{from } 1 \textbf{ to } n \textbf{ do } p\big\} \\ \textit{mezery} &= \textbf{proc}\big(n : \text{Int}\big)\big\{\textbf{if } n \geq 0 \textbf{ then } \textit{repeat}\big(n, \textit{write}(\text{'}\ \text{'})\big) \\ &\qquad\qquad \textbf{else } \textit{repeat}\big(-n, \textit{write}(\text{'\texttt{\textbackslash 0x08}'})\big)\big\} \end{aligned}$$

Funkcionální abstrakce se provádí nad výrazem tím, že některé jeho (volné) proměnné se prohlásí za parametry výsledné funkce. Počet těchto parametrů určuje tzv. *aritu*.

Umožňuje-li jazyk pracovat s funkcemi vyšších řádů, lze všechny funkce (procedury) převést na unární.

Podobně *procedurální abstrakce* se provádí nad příkazem.

ML:	**val** *sqr* $=$ **fn** $x \Rightarrow x * x$
Haskell:	*sqr* $= \backslash x \rightarrow x * x$
Lisp:	$($**define** *sqr* $($**lambda** (x) $(* \; x \; x)))$
Pascal:	**function** *sqr* $(x$: real$)$: real; **begin** *sqr* := $x * x$ **end**
C:	float *sqr* $($float $x)$ $\{$**return** $x * x;\}$

Princip abstrakce lze zobecnit i na jiné syntaktické kategorie, než funkce (např. na typovou abstrakci apod.).

Proměnné

Rozdílný význam v různých paradigmatech:

Funkcionální, logické – *čisté proměnné*, proměnné v matematickém smyslu.

Proměnná označuje hodnotu (třebaže předem neurčenou). Slouží také k označení parametrů funkcí resp. relací.

Haskell: $\mathit{fact}\ n \ = \ \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}(n-1)$

Prolog: $\mathit{lessthan}(\mathit{succ}(X), \mathit{succ}(Y))$:- $\mathit{lessthan}(X, Y)$.

Imperativní – tzv. *přepisovatelné proměnné*.

Proměnná označuje paměťové místo. Teprve toto paměťové místo slouží jako (přechodné) úložiště hodnoty. Hodnota uložená v paměťovém místě se může měnit. Tím se mění stav výpočtu.

Ve většině imperativních jazyků (Pascal, C, Java, . . .) se však jménem proměnné označuje i hodnota uložená v odpovídajícím paměťovém místě (automatické dereferencování).

Pascal:	**begin** *read*$(\underline{n})$; $\underline{n}$:= $\underline{\underline{n}} + 1$; *write*$(\underline{\underline{n}})$ **end**
Algol68:	Ref Int $\underline{n}$; $\underline{n}$:= $\underline{\underline{n}} + 1$
ML:	$\underline{n}$: int ref; $\underline{n}$:= $!\underline{\underline{n}} + 1$

Přepisovatelné proměnné složených typů mohou mít tzv. selektivní změny komponent.

Datum $= \{r$: Int, m : String, d : Int$\}$

dnes : Ref Datum

dnes := $\{r = 2005; m =$ `"říjen"`$; d = 5\}$

dnes.d := *dnes*.$d + 1$

$\approx$

dnes := $\{r = (!$*dnes*$).r; m = (!$*dnes*$).m; d = (!$*dnes*$).d + 1\}$

Datum' $= \{r$: Ref Int, m : Ref String, d : Ref Int$\}$

dnes : Datum'

dnes $= \{r =$ *ref* $2005; m =$ *ref* `"říjen"`$; d =$ *ref* $5\}$

dnes.d := *dnes*.$d + 1$

$\approx$

dnes.d := !$($*dnes*.$d)$ $+ 1$

rdnes : Ref Datum’

rdnes := *dnes*

rdnes.d := *rdnes*.$d + 1$

$\approx$

(!*rdnes*).d := !((!*rdnes*).d) $+ 1$

Příkazy

Významem příkazu je *stavový transformátor* $s : S \longrightarrow S$

MVar	množina programových proměnných representujících paměťová místa (i persistentní data)
Val	množina hodnot, jež mohou být do těchto míst uloženy
$\sigma : \mathit{MVar} \longrightarrow \mathit{Val}$	stav
S	množina všech stavů

Je-li p příkaz, pak jeho stavový transformátor se značí $[\![p]\!] : S \longrightarrow S$.

Příkazy obsahující výrazy bez vedlejších efektů

Předpokládejme pro začátek, že pouze příkazy mění stav, výrazy stav nemění.

Prázdný příkaz **skip**

$$[\![\textbf{skip}]\!] = id, \quad [\![\textbf{skip}]\!]\sigma = \sigma \text{ pro všechny stavy } \sigma \in S$$

Přiřazovací příkaz v := e

$$[\![v := e]\!]\sigma = \sigma', \qquad \text{kde } \sigma'(v) = \mathcal{M}[\![e]\!]\sigma,$$

$$\forall x \in \mathit{MVar} - \{v\}.\ \sigma'(x) = \sigma(x)$$

$\mathcal{M}[\![e]\!]\sigma$ je hodnota výrazu e ve stavu σ

Sekvence **begin** $p_1; \ldots; p_k$ **end**

$$[\![\textbf{begin } p_1; \ldots; p_k \textbf{ end}]\!] = [\![p_k]\!] \circ \cdots \circ [\![p_1]\!]$$

tj. pro všechna $\sigma \in S$

$$[\![\textbf{begin } p_1; \ldots; p_k \textbf{ end}]\!]\sigma = [\![p_k]\!]([\![p_{k-1}]\!](\ldots([\![p_2]\!]([\![p_1]\!]\sigma))\ldots))$$

Podmíněný příkaz **if** e **then** p_1 **else** p_2

$$[\![\textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2]\!]\sigma = \begin{cases} [\![p_1]\!]\sigma, \text{ pokud } \mathcal{M}[\![e]\!]\sigma = \textit{true} \\ [\![p_2]\!]\sigma, \text{ pokud } \mathcal{M}[\![e]\!]\sigma = \textit{false} \\ \bot \text{ jinak} \end{cases}$$

Příkaz cyklu **while** e **do** p

$$[\![\textbf{while } e \textbf{ do } p]\!]\sigma = \begin{cases} \sigma, \text{ pokud } \mathcal{M}[\![e]\!]\sigma = \textit{false} \\ [\![\textbf{while } e \textbf{ do } p]\!]([\![p]\!]\sigma), \text{ pokud } \mathcal{M}[\![e]\!]\sigma = \textit{true} \\ \bot \text{ jinak} \end{cases}$$

Tvrzení: Pro každý výraz e a příkaz p platí

$$[\![\textbf{while } e \textbf{ do } p]\!] = [\![\textbf{if } e \textbf{ then begin } p; \textbf{while } e \textbf{ do } p \textbf{ end}]\!]$$

Důkaz dosazením do definice $[\![\_]\!]$.

Násobné větvení

$$\textbf{case } e \textbf{ of}$$

$$v_1 \to p_1$$

$$v_2 \to p_2$$

$$\vdots$$

$$v_k \to p_k$$

$$[\![\textbf{case } e \textbf{ of } v_1 \to p_1, \ldots, v_k \to p_k]\!]\sigma = [\![p_i]\!]\sigma, \text{ kde}$$

$$i = \min\{j | 1 \leq j \leq k, \mathcal{M}[\![v_j]\!]\sigma = \mathcal{M}[\![e]\!]\sigma\},$$

existuje-li toto minimum, jinak σ

Výrazy s vedlejšími efekty

Připustíme, aby příkazy (tj. výrazy typu Command) byly podvýrazy výrazů jiného typu.

Pak jsou všechny výrazy stavovými transformátory $[\![e]\!] : S \longrightarrow S$, ale kromě toho vracejí hodnotu $\mathcal{M}[\![e]\!] : S \longrightarrow \mathit{Val}$.

Například v C ve stavu $\sigma,\ \sigma(x) = 5$, je

$$\begin{aligned} [\![x\,{+}{+}]\!]\sigma &= \sigma', \quad \sigma'(x) = 6,\ \sigma'(y) = \sigma\ y \text{ pro } y \neq x \\ \mathcal{M}[\![x\,{+}{+}]\!]\sigma &= 5 \end{aligned}$$

$[\![\textbf{skip}]\!] = id$

$$[\![x := e]\!]\sigma = \sigma', \qquad x \in \mathit{MVar}$$
$$\sigma'(x) = \mathcal{M}[\![e]\!]\sigma$$
$$\sigma'(y) = ([\![e]\!]\sigma)(y) \text{ pro } y \in \mathit{MVar} - \{x\}$$

$$[\![e_l := e_r]\!]\sigma = \sigma', \qquad v = \mathcal{M}[\![e_l]\!]([\![e_r]\!]\sigma)$$
$$\sigma'(v) = \mathcal{M}[\![e_r]\!]\sigma$$
$$\sigma'(y) = ([\![e_l]\!]([\![e_r]\!]\sigma))(y) \text{ pro } y \in \mathit{MVar} - \{v\}$$

$[\![\textbf{begin } p_1; \ldots; p_k \textbf{ end}]\!] = [\![p_k]\!] \circ \cdots \circ [\![p_1]\!]$

$$[\![\textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2]\!]\sigma = \begin{cases} [\![p_1]\!]([\![e]\!]\sigma), & \text{pokud } \mathcal{M}[\![e]\!]\sigma = \textit{true} \\ [\![p_2]\!]([\![e]\!]\sigma), & \text{pokud } \mathcal{M}[\![e]\!]\sigma = \textit{false} \\ \bot & \text{jinak} \end{cases}$$

$$[\![\textbf{while } e \textbf{ do } p]\!]\sigma = \begin{cases} [\![e]\!]\sigma, & \text{pokud } \mathcal{M}[\![e]\!]\sigma = \textit{false} \\ [\![\textbf{while } e \textbf{ do } p]\!]([\![p]\!]([\![e]\!]\sigma)) & \text{, pokud } \mathcal{M}[\![e]\!]\sigma = \textit{true} \\ \bot & \text{jinak} \end{cases}$$

Řadicí příkazy

Skoky

Explicitní přenesení řízení výpočtu do jiné části programu. Obvyklé ve starších jazycích a jazycích nižší úrovně. Většina jazyků klade na použití skoků omezení (například je možný jen skok do stejného nebo nadřazeného bloku).

Přílišné používání skoků vede k nečitelnému kódu a těžko odhalitelným chybám.

Například **go to** v C

Úniky

Ukončují provádění složeného příkazu, který únikový příkaz obsahuje.

nejvnitřnější

n-tý nejvnitřnější **exit** n

nejvnitřnější určitého druhu (cykly, case) C: **continue**, **break**, Prolog: **!**

nejvnitřnější funkcionální či procedurální abstrakce C: **return**

celý program **halt**, Fortran: **stop**, sh: **exit**

Výjimky

Mohou být ošetřeny procedurou pro *zpracování* výjimky (tzv. handler). Výjimek může být více, různých typů, a pro každou může být definováno jiné zpracování; různé části programu mohou definovat různá zpracování stejné výjimky.

Není-li výjimka zpracována (ošetřena), je *šířena* (propagována) do příkazu nadřazeného.

Některé výjimky jsou zabudované a vyvolávané zabudovanými operacemi, jiné lze uživatelsky definovat a *vyvolat* zvláštním příkazem (ML: **raise**, Java: **throw**).

Oblast zpracování (scope) výjimky je vymezena programovým blokem nebo zvláštní syntaxí (**try**. . . **end**).

Volání funkcí a předávání parametrů

aplikace funkce (procedury) na argumenty – tzv. volání funkce (procedury).

Volání *hodnotou* – striktní vyhodnocení

Volání *jménem* – normální vyhodnocení

Volání *dle potřeby* – líné vyhodnocení (v referenčně transparentních jazycích).

Volání *odkazem* (referencí)

Volání *hodnotou*, *výsledkem*, *hodnotou-výsledkem*.

Definice funkce (procedury)

$$f \ (x_1, \ldots, x_n) \quad = \quad b$$

levá strana – *hlavička* — pravá strana – *tělo*

$x_1, \ldots, x_n$ jsou *formální parametry* — výraz nebo příkaz

Aplikace funkce (procedury) na argumenty

neboli *volání* funkce (procedury)

$$f \quad (a_1, \ldots, a_n)$$

argumenty (výrazy)

Volání hodnotou

Nechť definice funkce (procedury) je $f\ (x_1, \ldots, x_n) \ = \ b$

a její volání je $f\ (a_1, \ldots, a_n)$, kde $a_1, \ldots, a_n$ jsou výrazy.

Pak

$$\begin{aligned} [\![f(a_1, \ldots, a_n)]\!]\sigma &= [\![b]\!]\sigma' \\ \mathcal{M}[\![f(a_1, \ldots, a_n)]\!]\sigma &= \mathcal{M}[\![b]\!]\sigma' \end{aligned}$$

kde pro všechna $i,\ 1 \leq i \leq n$

$$\begin{aligned} \sigma_0 &= \sigma \\ \sigma_i &= [\![a_i]\!]\sigma_{i-1} \\ \sigma'(x_i) &= \mathcal{M}[\![a_i]\!]\sigma_{i-1} \\ \sigma'(y) &= \sigma_n(y) \text{ pro každé } y \notin \{x_1, \ldots, x_n\} \end{aligned}$$

Volání jménem

Nechť definice funkce f je

$$f(x_1, \ldots, x_n) = b$$

Pak

$$\begin{aligned} [\![f(a_1, \ldots, a_n)]\!]\sigma &= [\![b']\!]\sigma \\ \mathcal{M}[\![f(a_1, \ldots, a_n)]\!]\sigma &= \mathcal{M}[\![b']\!]\sigma \end{aligned}$$

kde b' vznikne z b současnou substitucí výrazů $a_1, \ldots, a_n$ za všechny (volné) výskyty formálních parametrů $x_1, \ldots, x_n$, tj.

$$b' = [a_1/x_1, \ldots, a_n/x_n]b$$

Volání jménem se implementuje pomocí nulárních (bezparametrických) funkcí pro každý skutečný parametr předávaný jménem – tzv. *thunks*.

Př.:

function *sum* $(i$: Ref Int, m : Int, n : Int, **name** x : Real$)$: Real

= **begin** s : Ref Real;

$s := 0$;

for $i := m$ **to** n **do** $s := (s) + x$;

return (s)

end;

k : Ref Int;

$\vdots$

$sum\,(k, 1, 10, 1/(k * (k + 1)))$

C:

```
double sum ( int *i, int m, int n, double (*x)() )
  {
    double s;
    s = 0;
    for (*i = m; *i <= n; (*i)++) { s = s + (*x)(); }
    return s;
  }
...
double thunk1 (void) { return 1/(k*(k+1)); }
...
sum (&k, 1, 10, &thunk1);
```

Volání „dle potřeby“ – líná aplikace

Varianta volání jménem. Argumenty se však vyhodnocují nejvýše jednou. Používá se jen u referenčně transparentních jazyků, tedy u jazyků bez vedlejších efektů.

Volání odkazem

Podobné volání hodnotou, ale skutečné parametry funkcí a procedur smějí být jen (adresovatelná) paměťová místa (Ref T) a *neprovádí* se jejich implicitní dereferencování.

Formální parametry jsou nepřepisovatelné proměnné typu „odkaz na hodnotu“ (Ref T) – např. v Pascalu.

Při simulaci pomocí volání hodnotou (např. v C) mohou být formální parametry přepisovatelné proměnné uchovávající přepisovatelné proměnné (Ref (Ref T)) – dereferencování je pak dvojnásobné.

Pascal:

```
var a, b : Real;
procedure swap (var x, y : Real);
   var z : Real;
   begin  z := x;
          x := y;
          y := z
   end;
   ⋮
swap (a, b)
```

```
a, b : Ref Real
x, y : Ref Real
z : Ref Real
z := (x)
x := (y)
y := (z)


swap (a, b)
```

Implicitnímu dereferencování při volání zabrání způsob předávání parametrů.

C:

```
double a,b;
void swap (double *x, double *y)
  { double z;
     z = *x;
    *x = *y;
    *y =  z;
  }
...
swap (&a,&b)
```

```
a,b : Ref Real
x,y : Ref (Ref Real)
  z : Ref Real
  z := (*x)
 *x := (*y)
 *y := (z)
```

Implicitnímu dereferencování při volání zabrání statický operátor & – hodnotou se předají adresy

Volání výsledkem

Skutečnými parametry mohou být jen výrazy typu Ref a, tj. adresovatelná paměťová místa (např. přepisovatelné proměnné, prvky přepisovatelných datových struktur . . .).

Je-li definice funkce (procedury)

$$f(\textbf{out}\ y_1, \ldots, y_n) = b$$

Pak

$$\begin{aligned} [\![f(w_1, \ldots, w_n)]\!]\sigma &= \sigma'' \\ \mathcal{M}[\![f(w_1, \ldots, w_n)]\!]\sigma &= \mathcal{M}[\![b]\!]\sigma_n \end{aligned}$$

kde pro všechna $i,\ 1 \leq i \leq n$

$$\begin{aligned} \sigma_i &= [\![w_i]\!]\sigma_{i-1}, \quad \sigma_0 = \sigma \\ v_i &= \mathcal{M}[\![w_i]\!]\sigma_{i-1} \\ \sigma' &= [\![b]\!]\sigma_n \\ \sigma''(v_i) &= \sigma'(y_i) \\ \sigma''(u) &= \sigma'(u) \ \text{pro}\ u \notin \{v_1, \ldots, v_n\} \end{aligned}$$

Volání hodnotou-výsledkem

Skutečnými parametry mohou být jen adresovatelná paměťová místa.

Je-li definice procedury (funkce) $f(\textbf{inout}\ z_1, \ldots, z_n) = b$

Pak

$$\begin{aligned} [\![f(w_1, \ldots, w_n)]\!]\sigma &= \sigma''' \\ \mathcal{M}[\![f(w_1, \ldots, w_n)]\!]\sigma &= \mathcal{M}[\![b]\!]\sigma' \end{aligned}$$

kde pro všechna $i,\ 1 \leq i \leq n$

$$\begin{aligned} \sigma_i &= [\![w_i]\!]\sigma_{i-1}, \quad \sigma_0 = \sigma \\ v_i &= \mathcal{M}[\![w_i]\!]\sigma_{i-1} \\ \sigma'(z_i) &= \sigma_i(v_i) \\ \sigma'(u) &= \sigma_n(u) \ \text{pro}\ u \notin \{z_1, \ldots, z_n\} \\ \sigma'' &= [\![b]\!]\sigma' \\ \sigma'''(v_i) &= \sigma''(z_i) \\ \sigma'''(u) &= \sigma''(u) \ \text{pro}\ u \notin \{v_1, \ldots, v_n\} \end{aligned}$$

Smíšené volání – hodnotou, výsledkem, hodnotou-výsledkem

Jazyky, které podporují volání hodnotou, výsledkem, hodnotou-výsledkem, zpravidla dovolují, aby jedna víceparametrická funkce (procedura) své parametry vyhodnocovala různými způsoby, podle předpisu v hlavičce funkce (procedury).

$$f(\textbf{in}\ x_1, \ldots, x_m, \textbf{out}\ y_1, \ldots, y_n, \textbf{inout}\ z_1, \ldots, z_r) = b$$

Volání odkazem a volání hodnotou-výsledkem mají různou sémantiku.

Když se během zpracování funkce či procedury mění obsah přepisovatelné proměnné určené formálním parametrem předávaným odkazem, mění se *současně* obsah přepisovatelné proměnné, která je skutečným argumentem při volání.

Naproti tomu, když se mění obsah přepisovatelné proměnné určené formálním parametrem předávaným hodnotou-výsledkem, dějí se tyto změny *pouze lokálně* a obsah skutečného argumentu se změní až v okamžiku návratu.

Tento rozdíl může mít (zejména při nečistém stylu programování) vliv na výsledný stav.

var a, b : Int;
function f (x : Int) : Int
{ x := 1;
  a := $a + 1$;
  return x;
}

a := 1;
b := $f(a)$;

Viditelnost

Viditelnost jazykových entit

Statická

Pro každou definici jazykové entity (konstanty, proměnné, typu, . . .) je tzv. statickou sémantikou určena oblast platnosti definice.

Blokové jazyky (Pascal, Modula, Algol68, . . .)

Stromová struktura.

Globální a lokální entity.

Modulární jazyky (C, Modula, . . .)

Moduly se vzájemně nevnořují, ale mohou být tvořeny blokem.

Třídy viditelnosti: *public* / *private*.

Objektové jazyky – z hlediska viditelnosti podobné modulárním, rozlišují třídy a objekty.

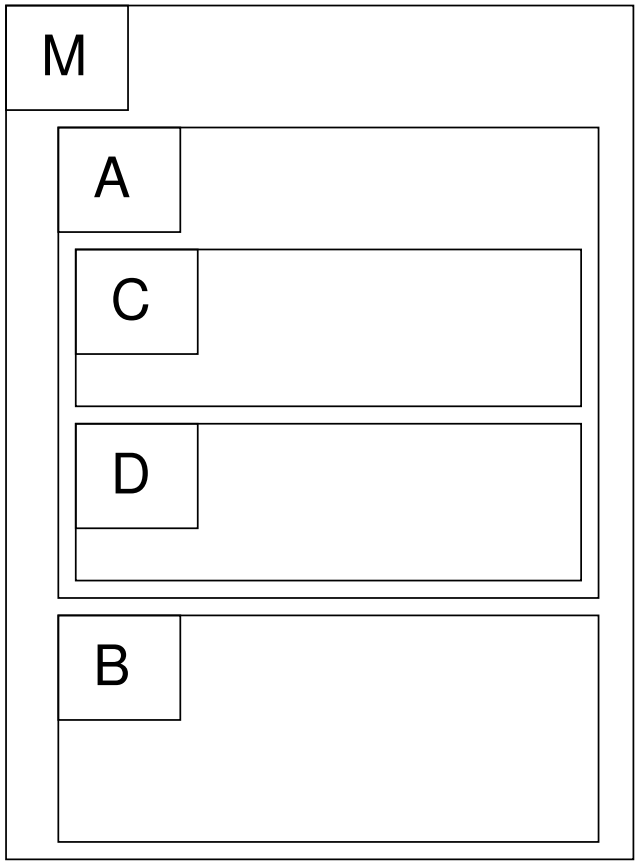
Amorfní jazyky (Basic)

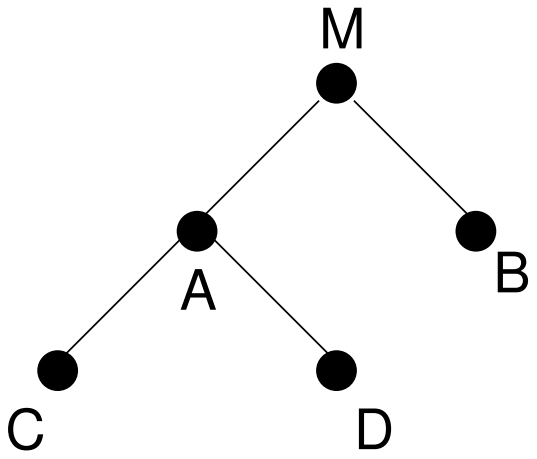
Nepoužitelné pro „programování ve velkém“.

Dynamická

Viditelnost jazykových entit závisí na momentálně aktivních programových jednotkách (blocích, funkcích, procedurách, . . .) v době běhu.

Téměř nepoužívaná (Lisp, Snobol, APL).





viditelnost

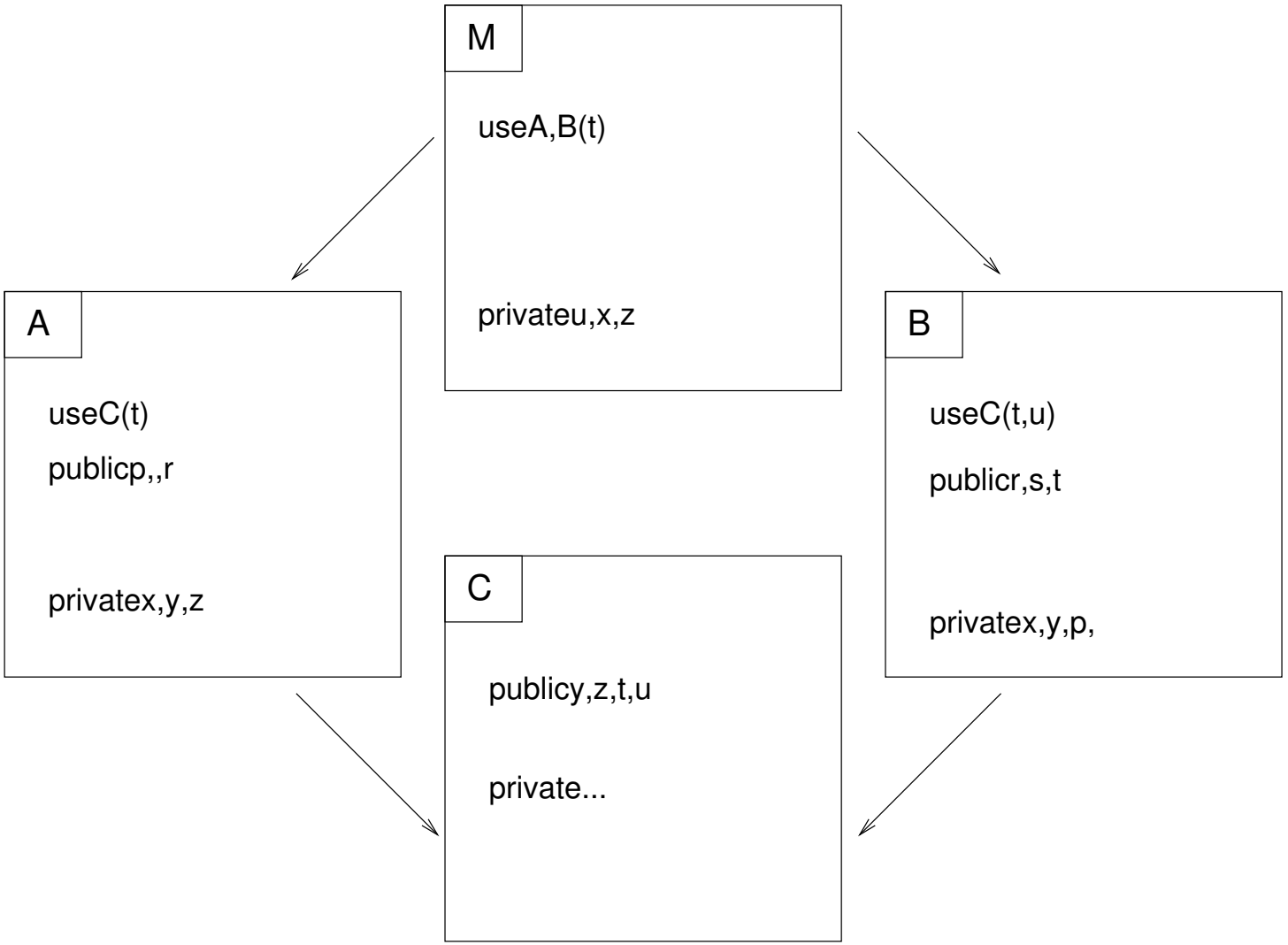
Oblast působnosti (scope) v jazycích s blokovou viditelností je podobná jako v matematickém zápisu:

$$\forall x \in A\, \exists f{:}A \to B.\ \ f(x) = b \ \wedge\ \forall x \in A.\, f(x) \leq b$$

Množina pojmenováných entit známých v dané oblasti působnosti zhruba odpovídá pojmu *jmenný prostor*. Avšak:

(i) některé jazyky mohou se jmennými prostory zacházet explicitně a nezávisle na jmenných prostorech implicitně určených bloky nebo moduly (C++);

(ii) jmenné prostory mohou být rozděleny na disjunktní části podle druhů pojmenovávaných entit – jména pro proměnné zvlášť, jména pro typy zvlášť, jména pro moduly zvlášť, . . . (Haskell).


M
useA,B(t)
privateu,x,z
A
useC(t)
publicp,,r
privatex,y,z
B
useC(t,u)
publicr,s,t
privatex,y,p,
C
publicy,z,t,u
private...

Dynamická viditelnost v Lispu

```
(define (f (lambda (x) (+ x y))))
(define (g (lambda (y) (* (f 2) y))))
(define (h (lambda (y) (+ (f y) (g (f 5))))))

(h 2)                                   y = 2
(+ (f 2) (g (f 5)))                             x = 2
(+ (+ 2 2) (g (f 5)))
(+ 4 (g (f 5)))                                 x = 5
(+ 4 (g (+ 5 2)))
(+ 4 (g 7))                                     y = 7
(+ 4 (* (f 2) 7))                                       x = 2
(+ 4 (* (+ 2 7) 7))
67
```

Omezování viditelnosti – zapouzdření

Moduly

Ada: **package** `ph` **is**

c: constant Float = 3e8

G: constant Float = 6.7e–11

h: constant Float = 6.6e–34

end `ph`

definuje uspořádanou trojici (c, G, h), ale její selektory jsou viditelné jen lokálně, tj. c, G, h jakožto funkce mají jednoprvkový definiční obor $\{\texttt{ph}\}$.

Zápis: `ph`.c, `ph`.G, `ph`.h.

Na rozdíl od agregátů (uspořádaných n-tic) mohou mít moduly i lokálně definované typy (popřípadě jiné jazykové entity).

Typy i data mohou být privátní (neexportovatelné) a veřejné (exportovatelné).

Třídy a objekty

Objekt

je modul se skrytými (privátními) přepisovatelnými proměnnými (tzv. *atributy*) a s veřejnými operacemi (tzv. *metodami*) nad těmito proměnnými .

Třída

(„generický objekt“) popisuje typ objektu.

Vlastní objekty se nazývaji *instance* třídy a vytvářejí se zvlášť:

- deklarací – statické objekty
- příkazem – dynamické objekty

Definice třídou exportovaných atributů a metod je součástí

- definice třídy – *statické* metody, případně atributy
- deklarace instance – atributy, někdy i metody

Dědičnost

mezi třídami je zavedena relace podtříd ($\leqslant\!:$).

Třída A je *podtřídou* třídy B (značíme $A \leqslant\!: B$), když dědí metody třídy B a případně přidává další.

Dědičnost je *jednoduchá*, má-li každá třída nejvýše jednu bezprostřední nadtřídu. Jinak je dědičnost *násobná*.

Vlastnosti OO jazyků

Třídy a objekty (statické nebo dynamické).

Dědičnost a inkluzní polymorfismus

- typový systém s podtypy
  $a : A,\ A \leqslant: A' \quad \Rightarrow \quad a : A'$

Viditelnost omezená na objekty a řízená pomocí public / private (příp. protected).

V (dynamických) objektech mohou být atributy i metody

- statické – definované ve třídách
- dynamické – definované v objektech

OO jazyky

Smalltalk čistě objektový, prototyp OO jazyků; netypovaný

Eiffel čistě objektový, typovaný

C++ původně jen rozšíření C pomocí makrojazyka cpp, není čistě objektové

Java spolu s C++ nejrozšířenější, není čistě objektová (má primitivní typy, které nejsou třídami), ale má čistší design než C++

Python, Ruby interpretovaný bytový kód

Modula 3, Ada 95, OCaML, ...

Správa paměti

Paměťové třídy dat

Persistentní data

existují nezávisle na výpočtu, jsou spravována souborovým podsystémem operačního systému. Obvykle jsou umístěna na vnějších paměťových médiích (soubory, databáze).

Transientní data

existují pouze po dobu výpočtu. Jsou obvykle umístěna v dostupné (procesem adresovatelné) části operační paměti (hodnoty proměnných, parametrů, obsahy přepisovatelných proměnných, . . .), ale i vně (dočasné soubory, data na portech, . . .).

V prostředí souběžného zpracování procesů je rozlišení dat na persistentní a transientní relativní.

Paměťová transientní data:

- Statická
- Automatická (zásobníková)
- Dynamická (haldová)

Správa paměti

Statická

prováděná překladačem (*statická data*, třída static, own).

[instrukce programu, systémová data, statická data, vyrovnávací paměti, . . .]

Zásobníková

v době výpočtu při zahájení/ukončení aktivace procedury, bloku, aplikace funkce na argumenty (*automatická data*, třída auto).

[lokální data v blocích, parametry funkcí a procedur, pomocné datové struktury ve funkcích, návratové adresy, . . .]

Dynamická

v době výpočtu, prováděná speciálními procedurami pro alokaci, dealokaci, přesouvání, scelování, . . . (*dynamická data*, třída dynamic).

1. řízená explicitně programem [malloc, free, new, dispose, mark, release]
2. spouštěná výhradně „run-time podporou“ výpočtu [garbage collecting, setřásání]

Fáze správy paměti

alokace (přidělení)

dealokace (uvolnění)

obnovení volné paměti může vyžadovat *gc*

scelování – změna polohy obsazených a volných bloků

- částečné
- úplné

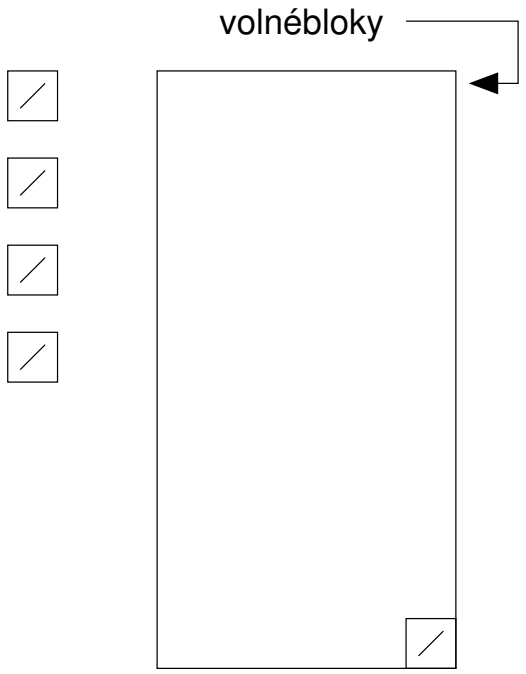
Problémy:

- smetí ($\Rightarrow$ gc)
- slepé odkazy
- fragmentace ($\Rightarrow$ částečné scelování)

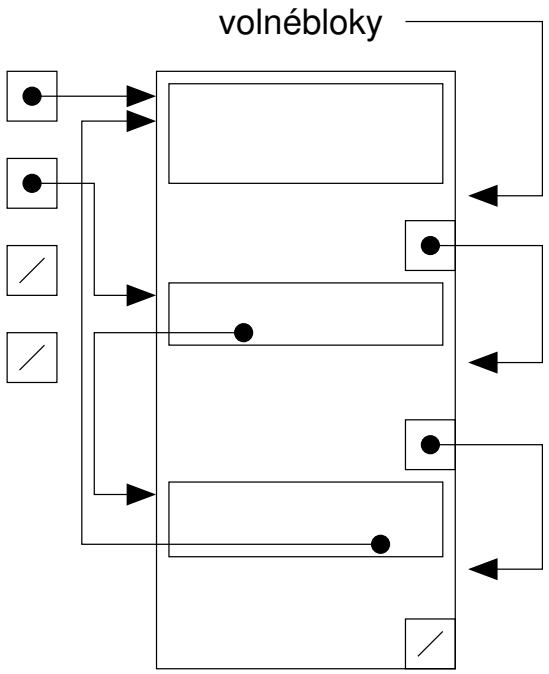
Správa dynamické paměti

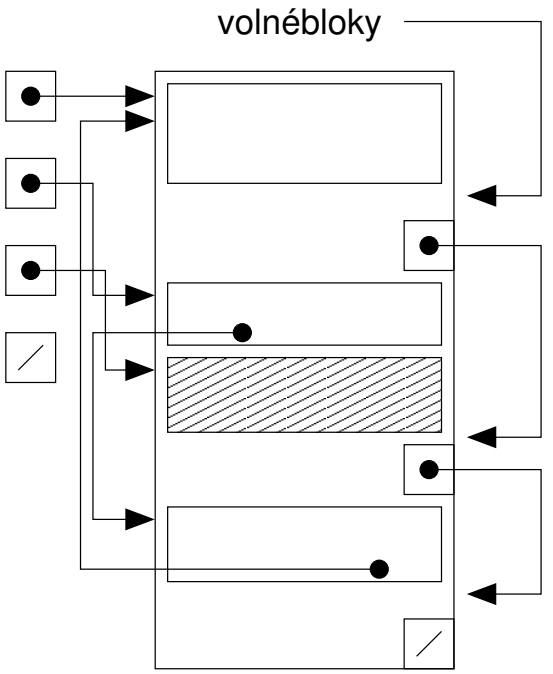
Inicializace

vytvoření seznamu volných bloků, seznamu mimohaldových ukazatelů



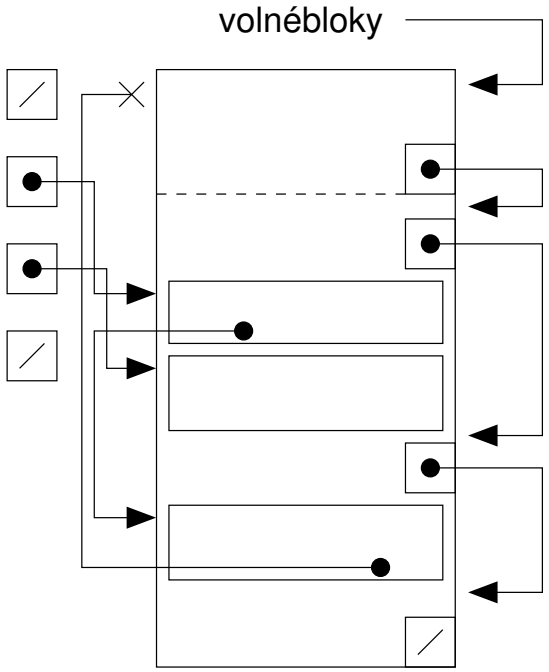
Alokace bloku požadované velikosti



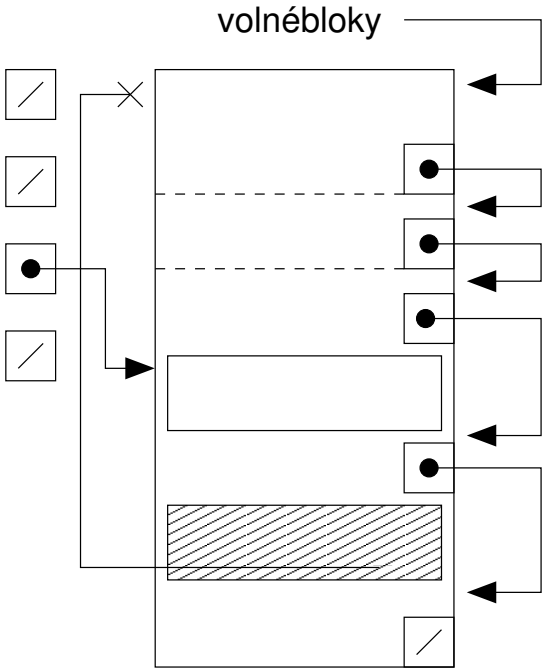


Uvolnění bloku

nekorektní – vznik slepého odkazu



nekorektní – vznik smetí

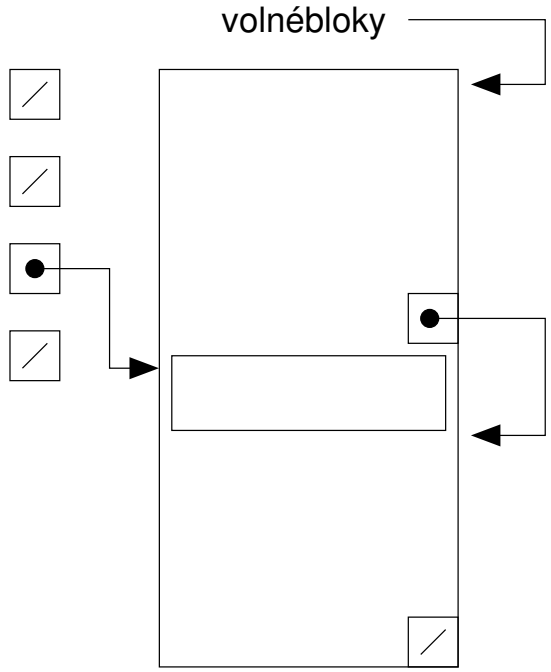


Úklid smetí (garbage collection)

- každý alokovaný blok má příznak dostupnosti (1 bit)
- všechny příznaky dostupnosti se nastaví na „nedostupné“
- postupuje se od známých ukazatelů a všechny navštívené bloky se označí „dostupné“ (při návštěvě bloku již označeného za „dostupný“ se hlouběji nepokračuje)
- nakonec se všechny nedostupné bloky zařadí do seznamu volných

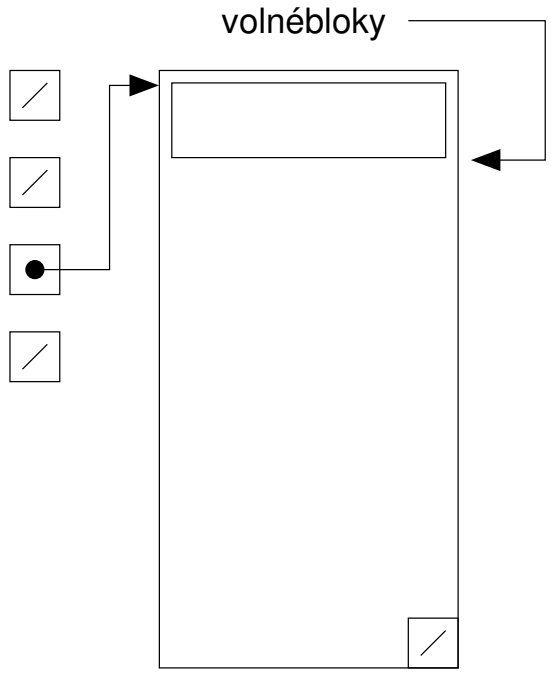
Scelování (spojování sousedních volných bloků)

je snazší, pokud se seznam volných bloků udržuje uspořádaný podle adres



Setřásání (úplné scelování)

způsobí změny ukazatelů



Paradigmata

Deklarativní paradigmata

Program popisuje, *co* je výsledkem.

Funkcionální – program je výraz a výpočet je jeho úprava.

Logické – program je teorie a výpočet je odvození z ní.

Imperativní paradigmata

Program popisuje, *jak* se dospěje k výsledku.

Procedurální – výpočet je provádění procedur nad daty.

Objektové – výpočet je předávání zpráv mezi objekty.

Logické paradigma

Patří mezi deklarativní paradigmata.

Logické programy popisují vztahy mezi hodnotami pomocí tzv. Hornových klausulí. Program se skládá ze seznamu klausulí (teorie) a z formule (cíle, dotazu). Výpočet je hledáním tzv. splňujících substitucí, tj. takových ohodnocení proměnných z cíle, při nichž cíl vyplývá (lze odvodit) z teorie.

Abstraktní syntax prototypového logického jazyka:

$$
\begin{aligned}
\text{Program} &::= \text{Klausule}^* \ \text{Formule} \\
\text{Klausule} &::= \text{Formule} \ \texttt{:-} \ \text{Formule}^* \\
\text{Formule} &::= \text{Pred} \ (\text{Term}^*) \\
\text{Term} &::= \text{Var} \mid \text{Fun} \ (\text{Term}^*)
\end{aligned}
$$

Pred, Fun, Var označují predikátové symboly, funkční symboly a proměnné.

Kontextová omezení (Statická sémantika)

Predikátové a funkční symboly mají konsistentní arity, tj.

$$\forall P \in \widehat{\mathsf{Pred}}\, \exists!\, a \in N\, \forall \varphi \in \widehat{\mathsf{Formule}}.\, \varphi = P(t_1, \ldots, t_n) \Rightarrow n = a$$

$$\forall F \in \widehat{\mathsf{Fun}}\, \exists!\, a \in N\, \forall t \in \widehat{\mathsf{Term}}.\, t = F(t_1, \ldots, t_n) \Rightarrow n = a$$

Rodič$(\mathit{Marie}, \mathit{Karel})$.

Rodič$(\mathit{Jan}, \mathit{Karel})$.

Rodič$(\mathit{Jan}, \mathit{Petr})$.

Rodič$(\mathit{Jana}, \mathit{Alena})$.

Rodič$(\mathit{Karel}, \mathit{Alena})$.

Sourozenec(x, y) :- *Rodič*(z, x), *Rodič*(z, y).

Potomek(x, y) :- *Rodič*(y, x).

Potomek(x, y) :- *Rodič*(y, z), *Potomek*(x, z).

Levá strana (tj. závěr) implikace se nazývá *hlava klausule*.

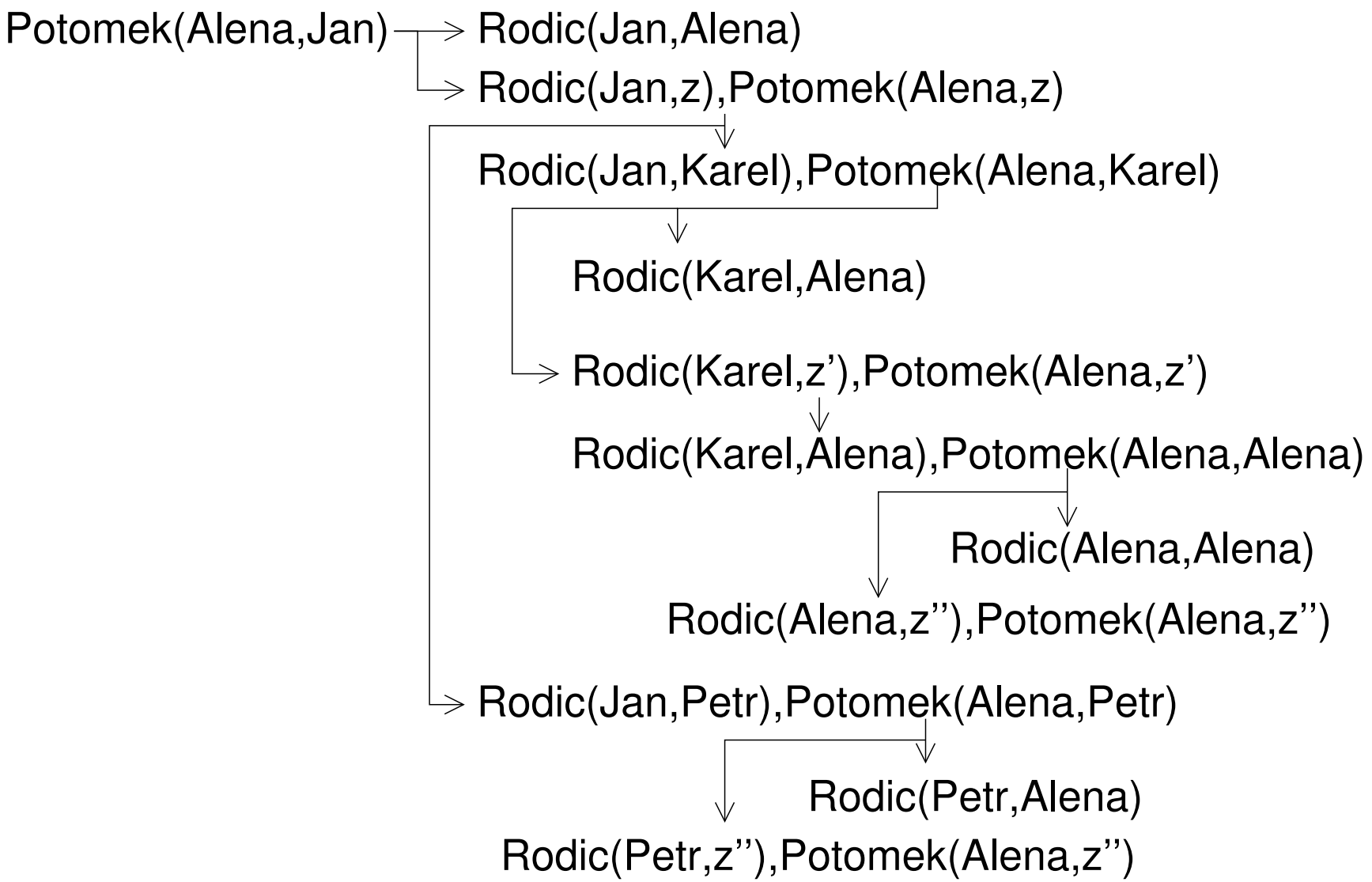
Předpoklady implikace se nazývají *podcíle*.

Klausule s nulovým počtem podcílů je tzv. *fakt*.

Proměnné v klausuli, které se vyskytují na levé straně klausule (v hlavě), jsou implicitně univerzálně kvantifikovány. Ostatní proměnné v klausuli (ty, jež se vyskytují jen na pravé straně – v podcílech) jsou implicitně existenčně kvantifikovány. (Proměnné jsou tedy v klausulích lokální!)

Např.

$$\forall x\, \forall y \;.\; \textit{Potomek}(x, y) \Leftarrow \exists z \;.\; \textit{Rodič}(y, z) \wedge \textit{Potomek}(x, z)$$


Potomek(Alena,Jan)
Rodic(Jan,Alena)
Rodic(Jan,z),Potomek(Alena,z)
Rodic(Jan,Karel),Potomek(Alena,Karel)
Rodic(Karel,Alena)
Rodic(Karel,z'),Potomek(Alena,z')
Rodic(Karel,Alena),Potomek(Alena,Alena)
Rodic(Alena,Alena)
Rodic(Alena,z''),Potomek(Alena,z'')
Rodic(Jan,Petr),Potomek(Alena,Petr)
Rodic(Petr,Alena)
Rodic(Petr,z''),Potomek(Alena,z'')

Termy

Zero – nulární funkční symbol (konstanta)

Succ – unární funkční symbol

$$\left.\begin{array}{l} \mathit{Zero} \\ \mathit{Succ}(\mathit{Zero}) \\ \mathit{Succ}(\mathit{Succ}(\mathit{Zero})) \\ \vdots \end{array}\right\} \text{uzavřené termy} \qquad \left.\begin{array}{l} \mathit{Succ}(x) \\ \mathit{Succ}(\mathit{Succ}(\mathit{Succ}(z))) \\ \vdots \end{array}\right\} \text{termy s proměnnými}$$

Klausule

$\mathit{LessThan}(\mathit{Zero}, \mathit{Succ}(y))$.

$\mathit{LessThan}(\mathit{Succ}(x), \mathit{Succ}(y))$:- $\mathit{LessThan}(x, y)$.

Dotaz

$\mathit{LessThan}(x, \mathit{Succ}(\mathit{Succ}(\mathit{Zero})))$.

Příklad: ternární relace spojování seznamů

$\mathit{Append}(\mathit{Nil}, y, y).$

$\mathit{Append}(\mathit{Cons}(x, s), t, \mathit{Cons}(x, u))$:- $\mathit{Append}(s, t, u).$

$\mathit{Append}(\mathit{Nil}, \mathit{Cons}(A, \mathit{Cons}(B, \mathit{Nil})), \mathit{Cons}(B, \mathit{Cons}(A, \mathit{Nil})))$

no.

$\mathit{Append}(\mathit{Cons}(A, \mathit{Nil}), \mathit{Nil}, \mathit{Cons}(A, \mathit{Nil}))$

yes.

$\mathit{Append}(\mathit{Nil}, y, y).$

$\mathit{Append}(\mathit{Cons}(x, s), t, \mathit{Cons}(x, u))$:- $\mathit{Append}(s, t, u).$

$\mathit{Append}(v, \mathit{Cons}(B, \mathit{Nil}), \mathit{Cons}(B, \mathit{Cons}(B, \mathit{Nil})))$

$v = \mathit{Cons}(B, \mathit{Nil}).$

$\mathit{Append}(\mathit{Cons}(A, \mathit{Nil}), w, \mathit{Nil})$

no.

$\mathit{Append}(v, \mathit{Cons}(B, \mathit{Nil}), \mathit{Cons}(A, \mathit{Cons}(x, \mathit{Nil})))$

$v = \mathit{Cons}(A, \mathit{Nil}), \quad x = B.$

$$\mathit{Append}(\mathit{Nil}, y, y).$$

$$\mathit{Append}(\mathit{Cons}(x, s), t, \mathit{Cons}(x, u)) \text{ :- } \mathit{Append}(s, t, u).$$

$$\mathit{Append}(v, w, \mathit{Cons}(A, \mathit{Cons}(B, \mathit{Nil})))$$

$$v = \mathit{Nil}, \quad w = \mathit{Cons}(A, \mathit{Cons}(B, \mathit{Nil}));$$

$$v = \mathit{Cons}(A, \mathit{Nil}), \quad w = \mathit{Cons}(B, \mathit{Nil});$$

$$v = \mathit{Cons}(A, \mathit{Cons}(B, \mathit{Nil})), \quad w = \mathit{Nil}.$$

Sémantika logického programu

Substituce

Subst je množina všech substitucí, tj. konečných zobrazení

$$\mathit{Var} \longrightarrow_{\mathrm{f}} \mathit{GTerm}$$

kde $\mathit{GTerm} = \{\tau \in \mathit{Term} \mid \tau \text{ neobsahuje proměnné}\}$ je množina všech tzv. *uzavřených* termů.

Každou substituci $\sigma : \mathit{Var} \longrightarrow \mathit{GTerm}$ lze jednoznačně homomorfně rozšířit na termy, $\bar{\sigma} : \mathit{Term} \longrightarrow \mathit{GTerm}$

$$\begin{aligned} \bar{\sigma}(v) &= \sigma(v) \\ \bar{\sigma}(g) &= g \\ \bar{\sigma}(f(t_1, \ldots, t_n)) &= f(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n)) \end{aligned}$$

Unifikátor σ formulí p, q je substituce taková, že $\bar{\sigma}(p) = \bar{\sigma}(q)$.

Sémantika

$$\begin{aligned} \mathcal{M}[\![\_]\!] &: \quad \textit{Programy} \longrightarrow \mathcal{P}(\textit{Subst}) \\ \mathcal{M}[\![(c,q)]\!] &= \{\sigma \in \textit{Subst} \mid c \vdash \bar{\sigma}(q),\ \forall D \subsetneqq \mathrm{Dom}\,\sigma.\ c \nvdash \overline{\sigma|_D}(q)\} \end{aligned}$$

Přitom $\vdash$ je relace tzv. odvoditelnosti cíle definovaná:

$$\begin{aligned} c = [\quad & f_{1,0} \text{ :- } f_{1,1}, \ldots, f_{1,m_1} \\ & \vdots \\ & f_{n,0} \text{ :- } f_{n,1}, \ldots, f_{n,m_n} \quad] \vdash q \end{aligned}$$

$$\Leftrightarrow$$

$$\exists \sigma \in \textit{Subst}\ \exists i, 1 \leq i \leq n.\ \bar{\sigma}(q) = \bar{\sigma}(f_{i,0}) \wedge \forall j, 1 \leq j \leq m_i.\ c \vdash \bar{\sigma}(f_{i,j})$$

Výsledkem výpočtu je *množina substitucí*.

$$\mathcal{M}[\![p]\!] = \emptyset \qquad \ldots\ldots \quad \text{„}\textit{No}.\text{“}$$

$$\mathcal{M}[\![p]\!] = \{\bot\} \qquad \ldots\ldots \quad \text{„}\textit{Yes}.\text{“}$$

Příklad: Ze sbírky hádanek pro chytré děti:

Jeden pes zje za jednu hodinu jednu kost'. Koľko psov zje za koľko hodín koľko kostí?

Správnou odpovědí je množina substitucí za „*koľko*“, „*koľko*“ a „*koľko*“

(zde nekonečná, aby s ní děti nebyly příliš rychle hotovy).

Logické jazyky s (positivními) Hornovými klausulemi implicitně předpokládají úplnost teorie (tj. nelze-li dotaz z teorie odvodit, pak je „*No*“) – tzv. „closed world assumption“.

Řízení výpočtu

Ve většině logických jazyků záleží na pořadí klausulí a na pořadí podcílů v klausuli (prostor substitucí se prohledává systematicky, takže i když řešení existuje, nemusí se při nevhodném pořadí klausulí najít, anebo když řešení neexistuje, nemusí se to v konečném čase zjistit).

Operátor ! (řez) v Prologu je řídicím operátorem, který zabraňuje opakovanému vyhodnocování podcíle, jestliže tento pocíl již jednou uspěl.

$$f_0 \text{ :- } f_1, !, f_2.$$

Pokud f_1 uspěje a f_2 následně neuspěje, nehledá se už další úspěšná substituce pro f_1.

$$f_0 \text{ :- } !.$$

Pokud se najde unifikátor pro f_0, nezkouší se s cílem unifikovat jiná klausule.

Gödel

```
MODULE      GCD.
IMPORT      Integers.
PREDICATE   Gcd : Integer * Integer * Integer.
Gcd(i,j,d) <-
            CommonDivisor(i,j,d) &
            ~ SOME [e] (CommonDivisor(i,j,e) & e > d).
PREDICATE   CommonDivisor : Integer * Integer * Integer.
CommonDivisor(i,j,d) <-
            IF (i = 0 \/ j = 0)
            THEN
              d = Max(Abs(i),Abs(j))
            ELSE
              1 =< d =< Min(Abs(i),Abs(j)) &
              i Mod d = 0 &
              j Mod d = 0.
```

Logické jazyky

Prolog

- 70. léta, Marseille, Edinburgh
- netypovaný jazyk
- plochá struktura a viditelnost:
  - všechny predikáty jsou globální
  - všechny proměnné jsou lokální (v klauzuli)
- není čistě logický
- nejrozšířenější logický jazyk (např. na FI *Sicstus Prolog*)

Gödel

- 90. léta, Bristol (J. W. Lloyd)
- typovaný jazyk
- modulární struktura
- parametrický polymorfismus
- P. M. Hill, J. W. Lloyd : The Gödel Programming Language, MIT Press 1994

Mercury

- multiparadigmatický jazyk (logický + funkcionální)
- polovina 90. let
- `http://www.cs.mu.oz.au/research/mercury/`

Aplikace logického programování

- zpracování přirozeného jazyka
- expertní systémy
- symbolická manipulace s výrazy, řešení rovnic
- simulace

Výhody

- vyšší úroveň, tj. program je bližší popisu problému než implementaci
- logika programu oddělena od řízení výpočtu
- snazší důkazy korektnosti

Nevýhody

- efektivita
- nevýhodné pro aplikace s intenzivním I/O

Funkcionální paradigma

Patří k deklarativním paradigmatům.

Deklarativní paradigmata — funkcionální, logické, . . .

Imperativní paradigmata — procedurální, objektové, souběžné, . . .

Rysy funkcionálních jazyků

- Nerozlišuje stavy, výpočet je jednostavový $\Rightarrow$ nemá selektory složek stavu, tj. přepisovatelné proměnné.
- jazyky vycházejí z lambda kalkulu nebo kombinátorového kalkulu, tedy pracují především s funkcemi a funkce jsou hodnotami „první kategorie“ $\Rightarrow$ vysoká míra ortogonality.

- bohatý typový systém – parametrický polymorfismus, typové a konstruktorové třídy
- uživatelské typy
- definice podle vzoru
- funkce vyššího řádu, kombinátory
- líné vyhodnocování, nekonečné datové struktury

Sémantika funkcionálního jazyka

Nepracuje se se stavy.

Hodnotový kontext (prostředí)

$$\varepsilon : \mathit{Var} \longrightarrow \mathit{Val}$$

Env ... množina všech hodnotových kontextů

Sémantická funkce

$$\mathcal{M} : \mathit{Term} \times \mathit{Env} \longrightarrow \mathit{Val}$$

Val ... sémantická doména

v netypovaných jazycích tvoří jeden CPO (complete partial order)

v typovaných jazycích systém CPO

CPO

je uspořádaná množina, v níž má každý nejvýše spočetný řetězec supremum.

monotónní funkce

f je monotónní, právě když pro každé dva prvky x, y takové, že $x \sqsubseteq y$, platí
$f(x) \sqsubseteq f(y)$

spojité funkce

f je spojitá, právě když pro každý spočetný řetězec $x_1, x_2, \ldots$ platí
$f\left(\bigsqcup(x_1, x_2, \ldots)\right) = \bigsqcup(f(x_1), f(x_2), \ldots)$

Každá spojitá funkce je monotónní.

Sémantické domény

Primitivní, tzv. *ploché*: Unit, Bool, Nat, Integer, Char

Operace nad doménami

Lift _	přidání nejmenšího prvku $\perp$
_ $\times$ _	kartézský součin
_ $+$ _	disjunktní sjednocení
_ $\rightarrow$ _	mocnina (doména spojitých funkcí)

Uspořádání na doménách

$$\forall x \in \mathsf{Lift}\ D\ .\ \bot \sqsubseteq_{\mathsf{Lift}\ D} x$$

$$\forall x, y \in D\ .\ (x \sqsubseteq_D y \Leftrightarrow x \sqsubseteq_{\mathsf{Lift}\ D} y)$$

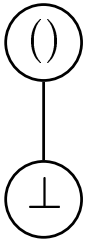
$$\forall x_1, y_1 \in D_1\ \forall x_2, y_2 \in D_2\ .$$
$$(x_1 \sqsubseteq_{D_1} y_1 \wedge x_2 \sqsubseteq_{D_2} y_2 \Leftrightarrow (x_1, x_2) \sqsubseteq_{D_1 \times D_2} (y_1, y_2))$$

$$\forall x_1, y_1 \in D_1\ \forall x_2, y_2 \in D_2\ .$$
$$((x_1 \sqsubseteq_{D_1} y_1 \Leftrightarrow x_1 \sqsubseteq_{D_1+D_2} y_1) \wedge (x_2 \sqsubseteq_{D_2} y_2 \Leftrightarrow x_2 \sqsubseteq_{D_1+D_2} y_2))$$

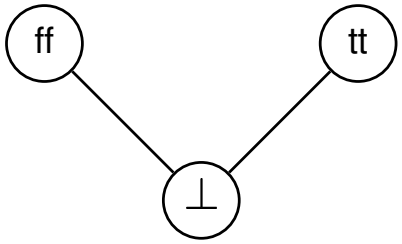
$$\forall f, g \in D_1 \to D_2\ .\ (f \sqsubseteq_{D_1 \to D_2} g \Leftrightarrow \forall x \in D_1\ .\ f(x) \sqsubseteq_{D_2} g(x))$$

Unit

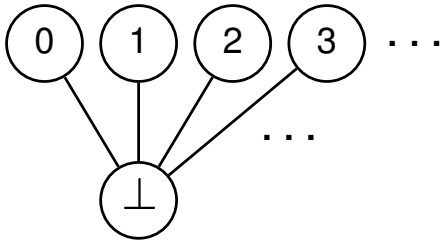
$\text{Unit}_\perp = \text{Lift Unit}$



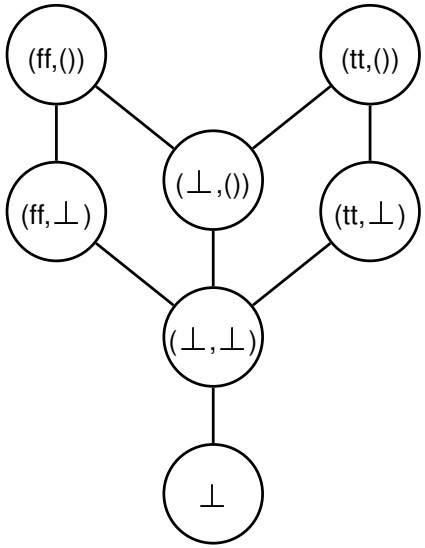
$\text{Bool}_\perp$



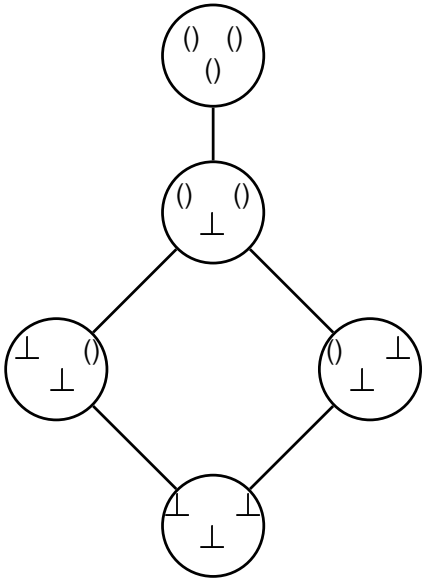
$\text{Nat}_\perp$



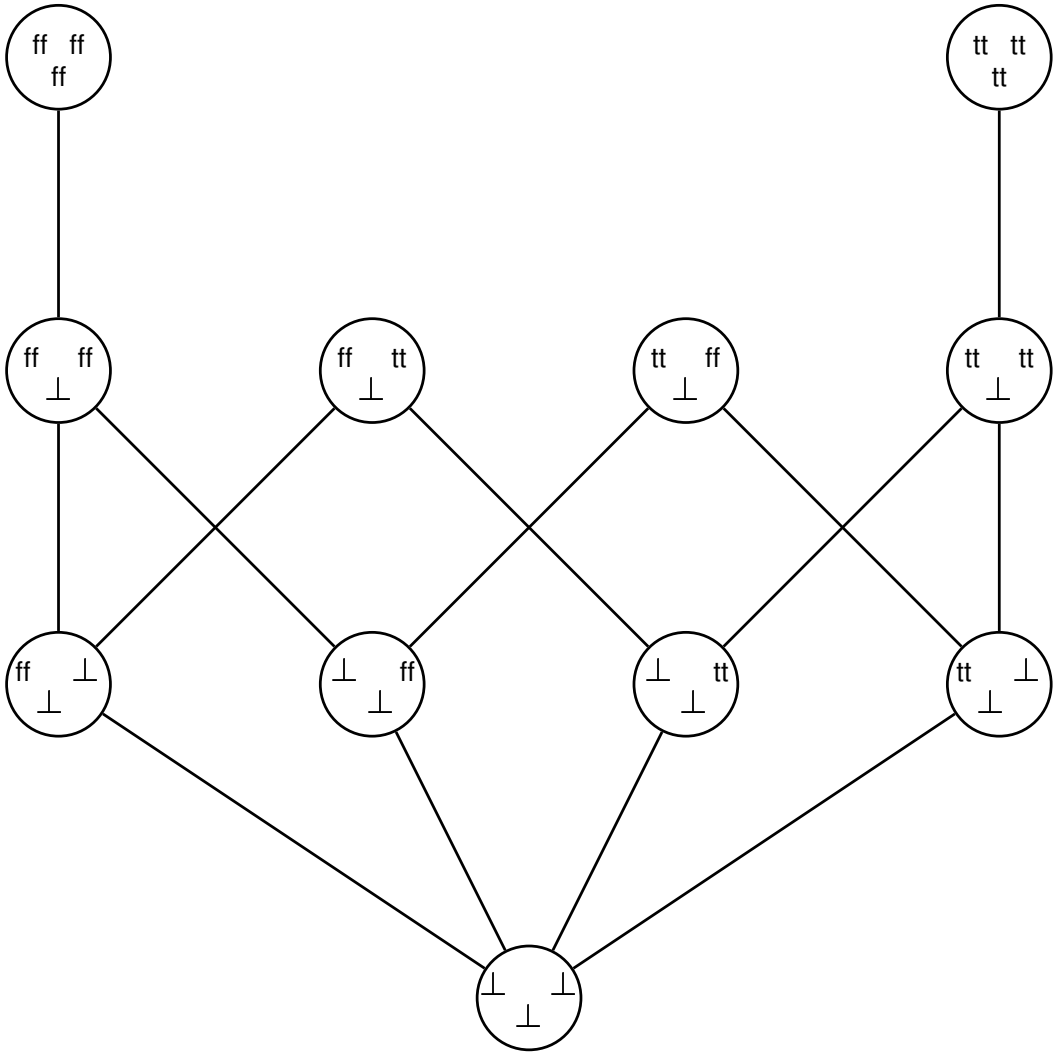
$\mathsf{Lift}\,(\mathsf{Bool}_\bot \times \mathsf{Unit}_\bot)$



$\mathsf{Bool}_\bot \to \mathsf{Unit}_\bot$



$\text{Bool}_\bot \to \text{Bool}_\bot$



Sémantika konstant

nezávisí na hodnotovém kontextu ε.

Pro libovolné $\varepsilon \in \mathit{Env}$

$$\begin{array}{rcl} \mathcal{M}[\![\mathit{True}]\!]\varepsilon & = & tt \\ \mathcal{M}[\![\mathit{False}]\!]\varepsilon & = & ff \\ \mathcal{M}[\![\mathit{0}]\!]\varepsilon & = & 0 \\ & \vdots & \end{array} \qquad \begin{array}{rcl} \mathcal{M}[\![\mathit{succ}]\!]\varepsilon & : & \mathit{Val} \to \mathit{Val}, \ldots \\ \mathcal{M}[\![\mathit{iszero}]\!]\varepsilon & & \ldots \\ \mathcal{M}[\![\mathit{not}]\!]\varepsilon & & \ldots \\ & \vdots & \end{array}$$

Sémantika výrazů

$$\mathcal{M}[\![x]\!]\varepsilon = \varepsilon(x)$$

$$\mathcal{M}[\![f\ e]\!]\varepsilon = (\mathcal{M}[\![f]\!]\varepsilon)(\mathcal{M}[\![e]\!]\varepsilon) \qquad \text{[líná aplikace]}$$

$$\mathcal{M}[\![g\ e]\!]\varepsilon = \begin{cases} \bot, & \text{když } \mathcal{M}[\![e]\!]\varepsilon = \bot \\ (\mathcal{M}[\![g]\!]\varepsilon)(\mathcal{M}[\![e]\!]\varepsilon) & \text{jinak} \end{cases} \qquad \text{[striktní aplikace]}$$

$$\mathcal{M}[\![\lambda x.\, e]\!]\varepsilon = f \text{ taková funkce z } \mathit{Val} \to \mathit{Val},$$

$$\text{že } \forall t \in \mathit{Val}.\ f(t) = \mathcal{M}[\![e]\!](\varepsilon[x \mapsto t])$$

$$\mathcal{M}[\![\textbf{if } e \textbf{ then } e' \textbf{ else } e'']\!]\varepsilon = \begin{cases} \mathcal{M}[\![e']\!]\varepsilon, & \text{když } \mathcal{M}[\![e]\!]\varepsilon = tt \\ \mathcal{M}[\![e'']\!]\varepsilon, & \text{když } \mathcal{M}[\![e]\!]\varepsilon = ff \\ \bot & \text{jinak} \end{cases}$$

Sémantika nerekursivních definic

$$\mathcal{M}[\![\textbf{let } v = e_1 \textbf{ in } e]\!]\varepsilon = \mathcal{M}[\![(\lambda v.\, e)\, e_1]\!]\varepsilon = \mathcal{M}[\![e]\!](\varepsilon[v \mapsto \mathcal{M}[\![e_1]\!]\varepsilon])$$

$$\begin{aligned}\mathcal{M}[\![\textbf{let } v_1 = e_1\,, \ldots,\, v_n = e_n \textbf{ in } e]\!]\varepsilon \;&= \\ \mathcal{M}[\![(\lambda v_1 \lambda v_2 \ldots \lambda v_n.\, e)\, e_1\; e_2\; \ldots\; e_n]\!]\varepsilon \;&= \\ \mathcal{M}[\![e]\!](\varepsilon[v_1 \mapsto \mathcal{M}[\![e_1]\!]\varepsilon,\, \ldots,\, v_n \mapsto \mathcal{M}[\![e_n]\!]\varepsilon])\end{aligned}$$

Sémantika rekursivních definic

Pro funkcionální jazyk s líným vyhodnocováním lze sémantiku rekursivních definic zavést takto:

$$\mathcal{M}[\![\textbf{letrec } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e]\!]\varepsilon = \mathcal{M}[\![e]\!]\varepsilon'$$

kde $\varepsilon' = \bigsqcup_{j \geq 0} \varepsilon_j, \quad \varepsilon_0 = \varepsilon[v_1 \mapsto \bot, \ldots, v_n \mapsto \bot]$

a pro každé $j \geq 0$ je $\varepsilon_{j+1} = \varepsilon_j\,[v_1 \mapsto \mathcal{M}[\![e_1]\!]\varepsilon_j, \ldots, v_n \mapsto \mathcal{M}[\![e_n]\!]\varepsilon_j]$.

Příklad

$\mathcal{M}[\![$ **letrec** $f = \lambda n.$ **if** $n ==$ **0 then** 1 **else** $n * f(n-1)$ **in** $f\,]\!]\varepsilon_0 = \mathcal{M}[\![f]\!]\varepsilon'$

$$
\begin{array}{lcl}
\varepsilon_0 & = & [f \mapsto f_0] \\
\varepsilon_1 & = & [f \mapsto f_1] \\
\varepsilon_2 & = & [f \mapsto f_2] \\
\varepsilon_3 & = & [f \mapsto f_3] \\
\varepsilon_4 & = & [f \mapsto f_4] \\
\vdots & & \\
\varepsilon' & = & [f \mapsto \bigsqcup f_j]
\end{array}
$$

	$\bot$	0	1	2	3	4	$\ldots$
f_0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$
f_1	$\bot$	1	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$
f_2	$\bot$	1	1	$\bot$	$\bot$	$\bot$	$\ldots$
f_3	$\bot$	1	1	2	$\bot$	$\bot$	$\ldots$
f_4	$\bot$	1	1	2	6	$\bot$	$\ldots$
$\vdots$							

Poznámka

Sémantiku rekursivních definic dostaneme zadarmo, zavedeme-li tzv. *kombinátor pevného bodu* **Y**. Potom lze rekursivní definici $v = e$ považovat za syntaktickou zkratku pro nerekursivní definici $v = \mathbf{Y}(\lambda v.\, e)$.

Potom $\mathcal{M}[\![\textbf{letrec } v = e_1 \textbf{ in } e]\!]\varepsilon = \mathcal{M}[\![(\lambda v.\, e)(\mathbf{Y}(\lambda v.\, e_1))]\!]\varepsilon$.

Platí $Y\ f = \bigsqcup_{k \geq 0} f^k(\bot)$, (kde $Y = \mathcal{M}[\![\mathbf{Y}]\!]\varepsilon$, pro libovolné ε),

tedy $Y(\lambda v.\, e) = \ldots((\lambda v.\, e)((\lambda v.\, e)((\lambda v.\, e)((\lambda v.\, e)((\lambda v.\, e)\bot)))))\ldots$.

Pomocí kombinátoru pevného bodu se sémantika rekursivních definic zavádí také u jazyků se striktním vyhodnocováním.

Funkcionální jazyky

LISP (LISt Processing) – John McCarthy, konec 50. let

- jednoduchá syntax, stejná pro data i algoritmy – homoikonický jazyk
- není čistě funkcionální
- netypovaný
- dynamická viditelnost
- dialekty: AutoLisp, eLisp, ...

Scheme netypovaný jazyk vycházející z Lispu, 80. léta

- syntax podobná Lispu; statická viditelnost, striktní vyhodnocování, streamy

ML typovaný jazyk, konec 70. let, Edinburgh

- striktní vyhodnocování
- dialekty: CaML, OCaML

Erlang dynamicky typovaný jazyk se striktním vyhodnocováním

konstrukce pro podporu paralelního vyhodnocování

Hope, Clean, Orwell, Miranda konec 80. let, vesměs líně vyhodnocované

často používané ve výuce programování; čistě funkcionální

Opal 90. léta, čistě funkcionální, modulární, striktně vyhodnocovaný

Haskell čistě funkcionální, modulární, líně vyhodnocovaný

čisté začlenění imperativních konstrukcí do referenčně transparentního jazyka

pomocí monadických typů a operací

Cayenne, Agda, Epigram experimentální jazyky

velmi silný (tedy nerozhodnutelný) typový systém

Souběžné zpracování

Multitasking souběžné zpracování úloh na jednom počítači

- úlohy využívají různé prostředky (typicky pomalejší I/O zařízení)
- interleaving
- víceuživatelský režim
- systém správy paměti je přizpůsoben souběžnému zpracování více úloh (stránkování, swapping, . . .)

Distribuované zpracování rozdělení úlohy na nezávislé podúlohy, méně komunikace

- výpočetní shluky
- globálně distribuované úlohy

Mobilní výpočty stěhování výpočtu po síti

Víceprocesorové počítače

- několikaprocesorové (MIMD)
- mnohaprocesorové (teoretický model PRAM)
- procesorová pole (SIMD)

Interference a nezávislost

Procesy P, Q jsou *nezávislé*, když jejich paralelní kompozice $P \| Q$ je vnějšně deterministická. V opačném případě jde o procesy *interferující*.

Nezávislost procesů je obecně nerozhodnutelná.

Dostatečná podmínka nezávislosti: Jsou-li oblasti paměti, k nimž přistupují procesy P, Q, disjunktní, a množiny dalších prostředků využívaných oběma procesy jsou také disjunktní, jsou procesy nezávislé.

Příklad interferujících procesů:

$$\begin{array}{l} \{\, v := 1\,; \\ \quad v := v + 1\,\} \end{array} \quad \| \quad \begin{array}{l} \{\, v := 0\,; \\ \quad v := 3 * v\,\} \end{array}$$

Existuje 6 možných prolínání, každé s jinou finální hodnotou proměnné v: 0, 1, 2, 3, 4, 6.

Sémantiku paralelní kompozice interferujících procesů nelze popsat přímo pomocí sémantik jednotlivých procesů (sémantika paralelní kompozice není skladebná).

Příklad: Nechť σ je libovolný stav a označme:

$P_1 = \{i := 1\},\ P_2 = \{i := 2\},\ P_1' = \{i := 0\,;\ i := i+1\},\ \sigma_k = \sigma[i \mapsto k]$.

Platí

$$[\![P_1]\!]\sigma = [\![P_1']\!]\sigma = \{\sigma_1\}$$

ale

$$\begin{aligned} [\![P_1 \parallel P_2]\!]\sigma &= \{\sigma_1, \sigma_2\} \\ [\![P_1' \parallel P_2]\!]\sigma &= \{\sigma_1, \sigma_2, \sigma_3\} \end{aligned}$$

Tedy

$$[\![P_1 \parallel P_2]\!]\sigma \neq [\![P_1' \parallel P_2]\!]\sigma$$

Naopak pro nezávislé procesy P, Q a každý stav σ platí jednoduše

$$[\![P \parallel Q]\!]\sigma \;=\; \{\sigma[\tau, \rho] \mid \tau \in [\![P]\!]\sigma,\ \rho \in [\![Q]\!]\sigma\}$$

Od procesů se obvykle očekává, že budou kooperovat, tedy nemohou být nezávislé. Neřízená (živelná) interference je však nepraktická, protože neomezeně interferující procesy je extrémně těžké programovat, ladit a spravovat. (Může však mít smysl na hardwarové úrovni.)

Proto je vhodné interferenci omezit na přesně určená místa výpočtu a oblasti sdílené paměti.

Podmíněná kritická sekce

Přepisovatelné proměnné sdílené procesy musí být jako takové deklarovány (**shared var** v). Složený příkaz **region** v **do** P vymezuje kritickou sekci vzhledem ke sdílené proměnné v. Proces v kritické sekci může být pozastaven (a přístup ke sdílené proměnné uvolněn) na základě tzv. *await podmínek*.

type MessageBuffer = **record** *size*, *front*, *rear* : 0..*capacity*;
items : array 1..*capacity* **of** Mes
end;
shared var *buffer* : MessageBuffer;

```
procedure sendmes (newitem : Mes);
  begin region buffer do
        begin  await buffer.size < capacity;
               buffer.size := buffer.size + 1;
               buffer.rear := buffer.rear mod capacity + 1;
               buffer.items[buffer.rear] := newitem
        end
  end;

procedure receivemes (out olditem : Mes);
  begin region buffer do
        begin  await buffer.size > 0;
               buffer.size := buffer.size − 1;
               olditem := buffer.items[buffer.front];
               buffer.front := buffer.front mod capacity + 1
        end
  end
```

Semafory

Semafor je globální objekt přístupný pouze pomocí metod

seminit(s, k)

semwait(s)

semsignal(s)

Skládá se ze tří číselných hodnot:

i ... iniciální hodnota zadaná operací *seminit*(s, i)

w ... počet provedených operací *semwait*(s)

s ... počet provedených operací *semsignal*(s)

Invariant: $0 \leq w \leq s + i$.

Parent:

seminit$(\mathit{nonempty}, 0)$;

seminit$(\mathit{nonfull}, k)$

Sender:

semwait$(\mathit{nonfull})$;

send_data;

semsignal$(\mathit{nonempty})$

Receiver:

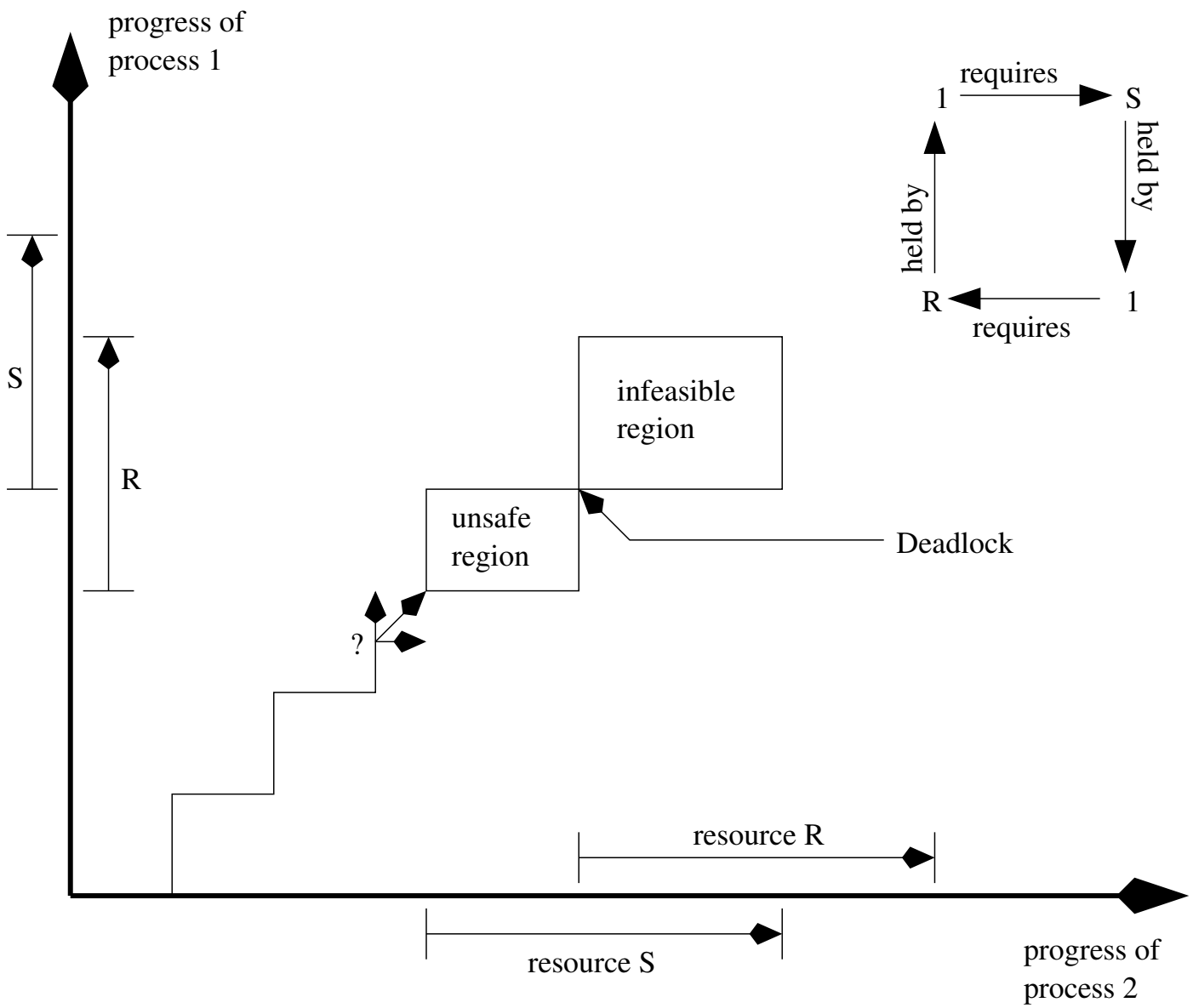
semwait$(\mathit{nonempty})$;

receive_data;

semsignal$(\mathit{nonfull})$

Problémy souběžného zpracování

- Nedeterminismus – při souběžném zpracování v mnohem větší míře než při sekvenčním; cílem je psát programy globálně (vnějšně) deterministické – chyby vzniklé porušením tohoto pravidla se obtížně detekují, protože běhy výpočtu jsou prakticky neopakovatelné.
- Závislost na relativní rychlosti zpracování.
- Zablokování – čekání procesů na sebe navzájem.
- Zmítání – „živá“ varianta zablokování (procesy tráví čas marnými pokusy na odstranění zablokování).
- Strádání – situace, kdy některé procesy „stojí“ (setrvávají ve stejném stavu) a ostatní běží na jejich úkor – je důsledkem tzv. nespravedlivého plánování.


progress of
process 1
requires
1
S
held by
held by
R
1
requires
S
R
infeasible
region
unsafe
region
Deadlock
?
resource R
resource S
progress of
process 2

Zablokování (deadlock)

je stav, kdy množina procesů čeká na přístup k prostředkům a z důvodu vzájemně nekompatibilních požadavků tento přístup nezíská.

K zablokování může dojít, právě když platí současně následující podmínky:

- Vzájemné vyloučení – každý prostředek je přidělen nejvýše jednomu procesu.
- „Wait & hold“ – procesy čekající na prostředky si drží jiné dříve přidělené prostředky.
- Cyklické čekání – v bipartitním grafu procesů a prostředků s hranami „chce“ a „je přidělen“ je cyklus.

Řešení zablokování

Neřešení – ignorování deadlocku až do zmrznutí OS, pak reboot.

Detekce a zotavení – rozpoznání, že došlo k deadlocku, a zabití procesu.

Prevence zrušením podmínky „wait & hold“ – vyžaduje se, aby každý proces požádal naráz o všechny prostředky, které bude potřebovat – plýtvání prostředky.

Prevence zrušením podmínky cyklického čekání – na prostředcích se zavede lineární uspořádání a vyžaduje se, aby byly požadovány jen v tomto pořadí.

Vyhýbání se zablokování – sledováním, zda se výpočet nepřiblížil nebezpečné oblasti.

„Živá“ varianta deadlocku – *livelock* (zmítání).

Algoritmus plánování přidělování prostředků procesům je *spravedlivý*, když zaručuje, že každý proces bude čekat na přístup k požadovaným prostředkům nejvýše konečně dlouho.