# 1) Sémantika: operační sémantika, denotační sémantika, pevný bod funkce, vázání jmen, stav a data programu.

Operační sémantika – sémantika malého kroku a velkého kroku. [TPJ přednáška 1]

## Sémantika malého kroku (SSOS)

**S = <CF, 🡺 FC, IF, OF>**

* Postupně se aplikují přepisovací pravidla

CF – doména konfigurací (obor hodnot)

🡺 – přepisovací relace (transformuje konfigurace)

FC – nezjednodušitelné konfigurace (zapisuje se přeškrtnutou šipkou)

IF – vstupní funkce

OF – výstupné funkce

(Možné konce – konečný stav, uváznutí – např. něco, na co nemáme pravidlo, nekonečná smyčka)

### CF = množina konfigurací

* Konfigurace = stav paměti + stav programu (paměť / program může být konstantní i nulový)
  + Konfigurace = přepisovací pravidla,
  + stav paměti = zobrazení „jméno proměnné“ 🡪 hodnota,
  + stav programu = seznam zbývajících přepisovacích pravidel
  + (VarName -> (Number U Boolean) ) × Program

### FC = množina finálních konfigurací

* Musíme říct, s čím skončíme
* Vybrat nějaké konfigurace, a když dojdu k nějaké z nich, tak jsem skončil
* Imperativní jazyk = FC jsou takové, které mají prázdný program

### => = přepisovací/přechodová relace

Konfigurace => konfigurace = přepisovací pravidla (o konfiguracích se mluví, je-li nějaká sémantika, jinak se říká stav)

(paměť, program) => (jiná/stejná paměť, jiný/stejný program)

Pokud už prvek nelze dále přepsat podle žádného pravidla, je v **normální formě. NF** jsou většinou finální konfigurace. ***obecně:*** NF = výsledek U zaseklý stav

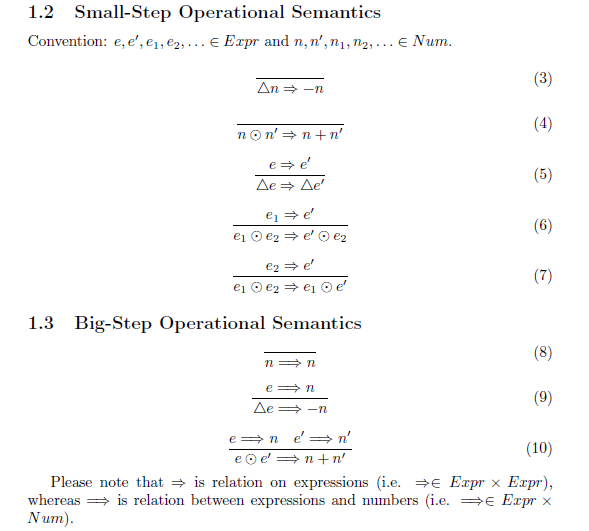
* Silně normalizující: tvoří normální formu v každé cestě
* Slabě normalizující: Alespoň v jedné cestě tvoří normální formu.

### IF = vstupní funkce

* Konverze dat od uživatele do počátečního stavu sémantiky
* Program × Input 🡪 CF
* IF(program) = ({}, program); {} = prázdná paměť
* Jiná syntaxe zápisu funkce:
  + IF = P **.** ({}, P)

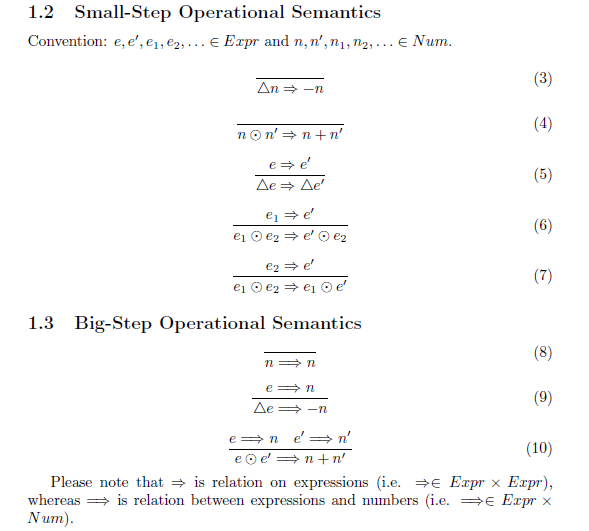
### OF = Výstupní funkce

* Konverze mezi finální konfigurací a výstupem programu



## Operační sémantika velkého kroku (Big-Step Operational Semantics)

* Ze vstupů přepíše hned (co nejrychleji) na výsledek



## Denotační sémantika

* Oproti operační sémantice se nezajímá o syntaxi (jak je to napsané – {,},(,),if, …), ale o sémantiku = jaký to má význam.

[[syntaktická algebra]] = sémantická algebra

[[3]] (dvojitá hran. závorka) = 3 (číslo se přepíše na číslo)

[[ /\ ]] = λx.-x (unární mínus)

[[ /\ E ]] překlápí se /\ a E zvláště: 🡪 [[ /\ ]] ( [[ E ]] ) – denotační sémantika trojúhelníka použitá na denotační sémantiku E. [[ /\ ]] je vlastně funkce, definice viz předchozí

Př.:

[[ /\ 5 ]] = [[ /\ ]] [[5]] = λx.-x (5) = -5

- přepíše se to na sémantiku trojúhelníku aplikovanou na sémantiku podvýrazu (= e)

## Pevný bod funkce

bod X, ve kterém platí F(X) = X.

* Využívá se pro rekurzivní funkce
* Y kombinátor v lambda kalkulu: Y = λf (λx . f(x x))(λx . f(x x))
* Př.: Generující funkce faktoriálu **fact = λF. λX.if x==0 then 1 else F(decrement(X))**
* Generující funkci dám do Y kombinátoru:
  + Y fact =

= λf (λx . f(x x))(λx . f(x x)) fact =

= (λx . fact(x x))(λx . fact(x x)) =

= fact ( (λx . fact(x x) (λx . fact(x x) ) =

= fact (Y fact)

* + Y fact je pevný bod funkce, která počítá faktoriál

## Vázání jmen

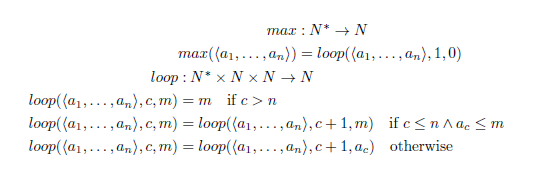
* Lambda kalkulus
* λX.λY.A B C X Y = λX.(λY.(A B C)) X Y
* X a Y jsou vázané proměnné (je před nima lambda), A, B, C jsou volné proměnné = globální proměnné
* Pokud vázanou proměnnou přejmenuji, tak se nic nestane.
* Funkce, co mají jen vázané proměnné, vrátí při každém zavolání stejný výsledek. Funkce s volnými proměnnými jsou závislé na globálním kontextu.
  + Funkce, co mají jen vázané proměnné, se v λkalkulu nazývají **kombinátory.**

## Stav a data programu

**Stav** = proměnné v prostředí, proměnné mají typ, prostředí = množina všech proměnných, kontext (v handoutech **Γ** (Gama))

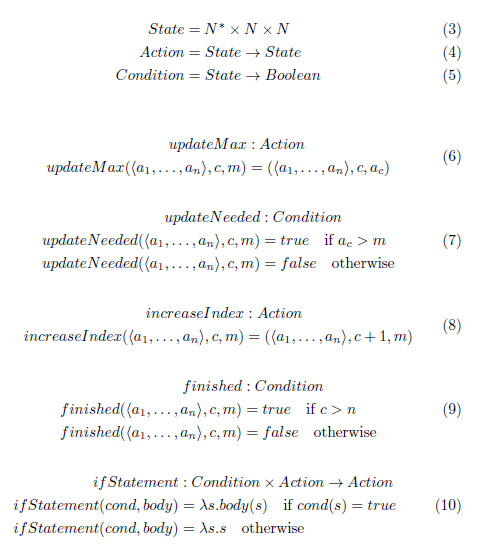
Čistě funkční jazyky (a matematika) jsou bezestavové, stavové výpočty mohou být reprezentovány jako iterace skrz stavy.

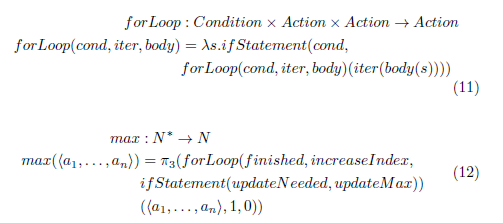
Př.: Funkce na nalezení maxima z pole:

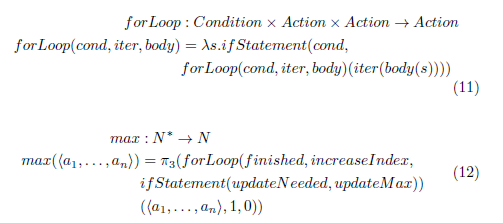


**Monády** – struktury, co reprezentují výpočet jako sekvenci kroků.

Př.: Nalezení maxima pomocí monád:







### Vyhodnování proměnných

- call by name, call by value, call by denotation

**Call by value**

* parametr funkce se vyhodnotí při zavolání funkce, vkládá se hodnota.

**Call by name**

* parametr se v okamžiku volání vloží jako nevyhodnocený výraz na místa, kde je použit.

**Call by denotation**

* parametr se dosadí až při vyhodnocení výrazu

### Data programu

* dělí se na součiny, sumy a sumy součinů

**Součiny:**

Positional data = N-tice, každý prvek může mít jiný typ

Sequence, List = pole

Named = třída

Nonstrict, stream = data, která se získají/vypočítají v okamžiku, kdy je potřeba (např. InputStream, odněkud se to vezme)

**Sumy:**

Union v C, nadtypy v Javě

**Sumy součinů:**

Binární a ternální operátory, double dispatch

# 2) Statická sémantika: typy, polymorfní typy, typy vyššího řádu, rekonstrukce (inference) typů, abstraktní typy.

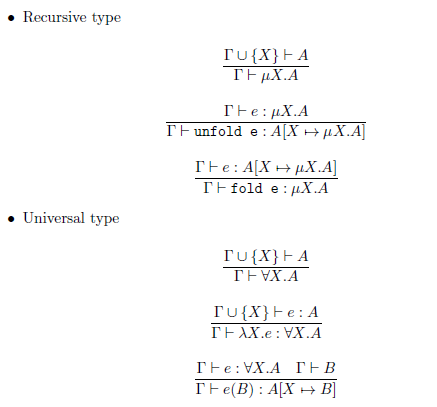
Statická sémantika je řešena při překladu programu, zde jsou definovány a deklarovány jednotlivá pravidla a prvky programovacího jazyka. V těchto prvcích je zahrnuta jazyková konstrukce, její typy parametrů, význam příkazů a další prvky. Statická sémantika dále kontroluje statické typy a práci s tabulkou definovaných programových symbolů.

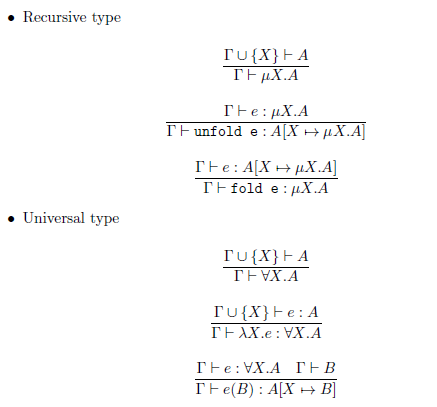
* Staticky tipované jazyky požadují uvedení datového typu u každé deklarace. Zde nelze deklarovat proměnnou, či funkci nebo objekt bez zadání datového typu.
* Všechny typové kontroly jsou prováděny staticky při překladu. Už při překladu má být každé proměnné přiřazen datový typ.
* Je možné daný datový typ přímo přetypovat. Přetypování především slouží k obcházení typových kontrol.
* Výhodou statického typování je lepší možnost odhalení typových chyb.
* Hlavní nevýhodou této metody je větší složitost programových konstrukcí, délka zdrojového kódu a tím i menší pružnost programovacího jazyka.
* Mezi neznámější zástupce staticky typovaných jazyků patří Java, Ada a jazyk C.

## Typy:

* Základní typy (např. Number, Boolean, String)
* Typ má každá proměnná, výraz, funkce
* Typ = množina přípustných hodnot a operací s nimi
* TopType
* **Type preservation** = zachování typů během přepisovací relace
* **Progress** = když mám nějaký term v konfiguraci, tak pak to je buď finální konfigurace, nebo lze ještě nejméně jednou přepsat
  + Dk. Analýzou pravidel
* Type preservation + progress = **soundness**. Soundness je dle Píšeho a Buka velmi důležitá vlastnost typů.
* **Terminace** = nadefinuji „energii“, nadefinuji, že při každém přepsání se musí snížit.
* **Determinismus** = pokud se výraz přepíše na hodnotu *v* a tentýž výraz jinou cestou na hodnotu *v’*, pak *v = v’*. Platí to, protože konfulence relace.
* **Konfluence** = „strom vyhodnocování relace se rozdělí a pak se zase spojí“

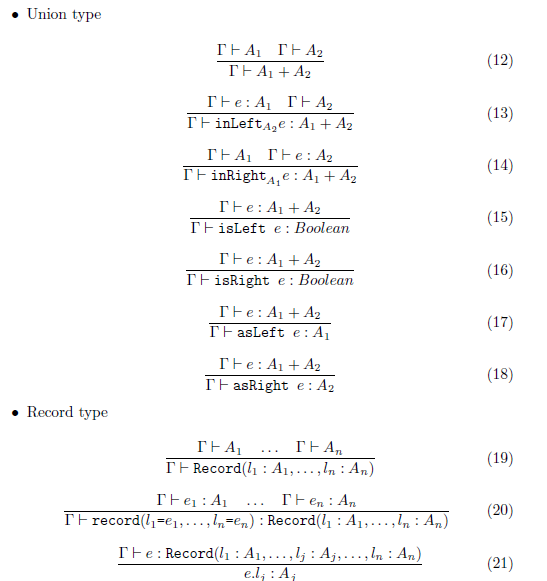
## Polymorfní typy

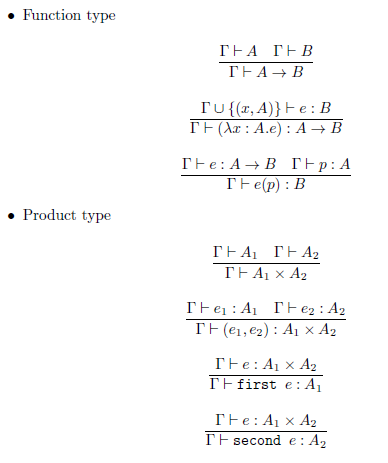




## Typy vyššího řádu

## 02-pokrocile typy 1.png





## 02-pokrocile typy 2.png

## Abstraktní typy

Fronta, HashMapa, Množina (Set), Seznam (List), Zásobník (Stack), graf, …

## Rekonstrukce (inference) typů

* Z proměnné vyvodit její typ, není to ve staticky typovaných jazycích (C++)