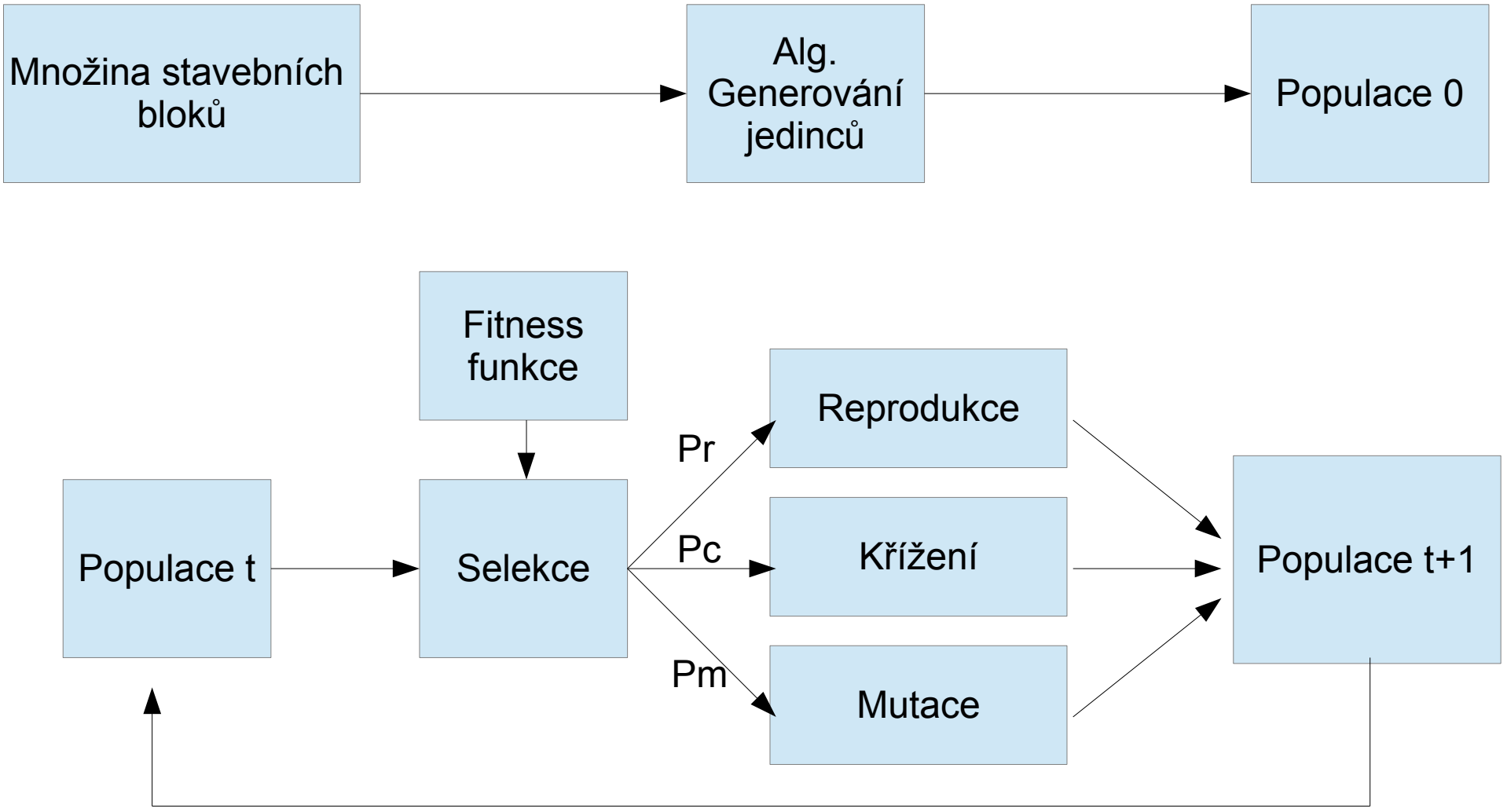
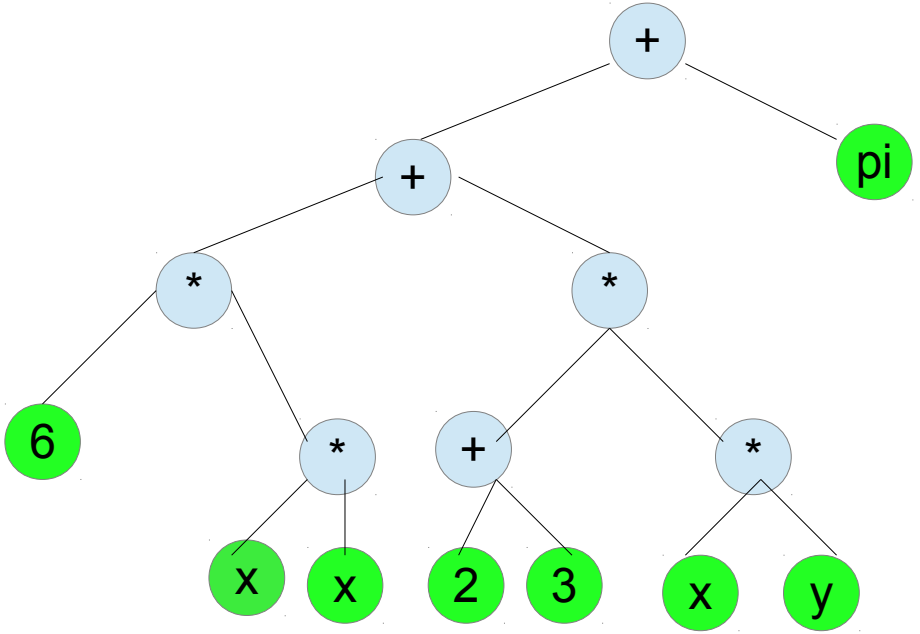
Genetické programování nad typovaným lambda kalkulem

Tomáš Křen


Množina stavebních bloků
Alg. Generování jedinců
Populace 0
Fitness funkce
Reprodukce
Pr
Populace t
Selekce
Pc
Křížení
Populace t+1
Pm
Mutace

Jak vypadá jedinec?

- Jedinec je **syntaktický strom** programu.
- Nelistové uzly jsou jména funkcí. (mn. **F**)
- Listové uzly jsou jména proměnných, konstanant, nebo konkrétních hodnot (mn. **T** = Terminály)
- Mn. stavebních bloků **Γ = T ∪ F**




```
function(x,y){
  return 6*x*x+(2+3)*x*y+pi ;}
```

Omezení jedince v klasickém GP

- T a F musejí být **nad jediným typem**:
  - Všechny prvky T jsou toho samého typu A (např. Int)
  - Všechny funkce mají typ tvaru $A \times A \times \ldots \times A \rightarrow A$
    - Tzn. mohou mít různé počty vstupů, ale všechny jsou typu A
    - Výstup je typu A

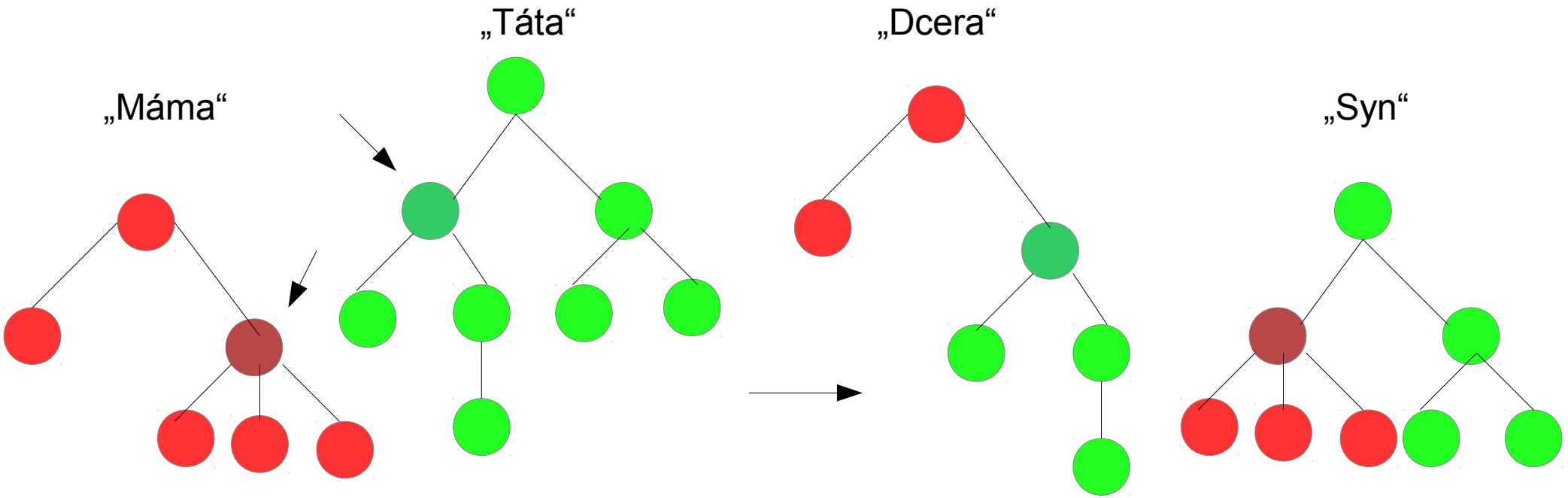
Jak se generuje počáteční populace?

Koza : *Ramped half-and-half*

- Nejdřív si hodíme mincí zda tento strom bude **„full“ nebo ne**
- Pak si hodíme kostkou čímž **určíme max. hloubku** stromu (tzn. Z {1,..,6})
- Do kořene (hloubka 0) stromu dáme náhodný prvek z mn. **F**
- Pokud jsme už v maximální hloubce, dám tam něco z mn. **T**
- Pro prostřední hloubky:
  - Pokud je tento strom „full“ : přidej náhodný prvek z **F**
  - Jinak : přidej náhodný prvek z **T** ∪ **F**

Jak se kříží?

V obou stromech (rodičích) vybereme náhodně libovolný uzel a podstromy s těmito uzly jako kořeny prohodíme.

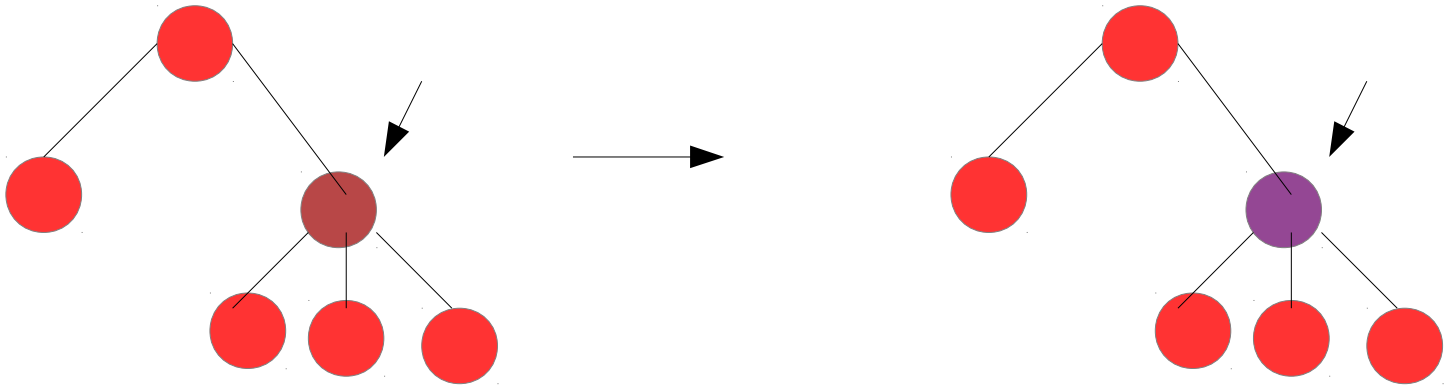


Jak se mutuje?

(Koza ani nemutuje.)

Náhodně vybereme uzel stromu a následně z Γ vybereme jiný uzel se stejným typem (tzn. se stejnou aritou), kterým vybraný uzel nahradíme.

(Jiná mutace : vygenerovat podstrom)



Nám se nelíbí omezení jediného typu

Abychom to vyřešili hezky obecně, zavoláme si na pomoc lambda kalkul.

Ten nám následně umožní zvolit si Γ libovolně.

Lambda kalkulus :
„nejjednoduší prog. jazyk na světě“

- Programu se v lambda kalkulu říká term.
- Term můžeme dostat 3 způsoby:

  (Nechť je **x** *nějaký řetězec písmen.)*

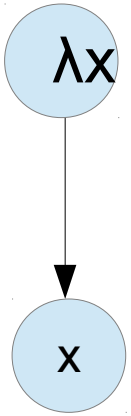
  - **x** je term. (= proměnná (nebo konstanta))
  - Pokud **M** je term → **(λx.M)** je term. (= λ abstrakce)
  - Pokud **M** a **N** jsou termy → **(M N)** je term (= aplikace funkce)

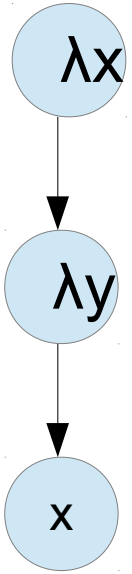
  - `function(x){return M;}`
  - `M(N)`

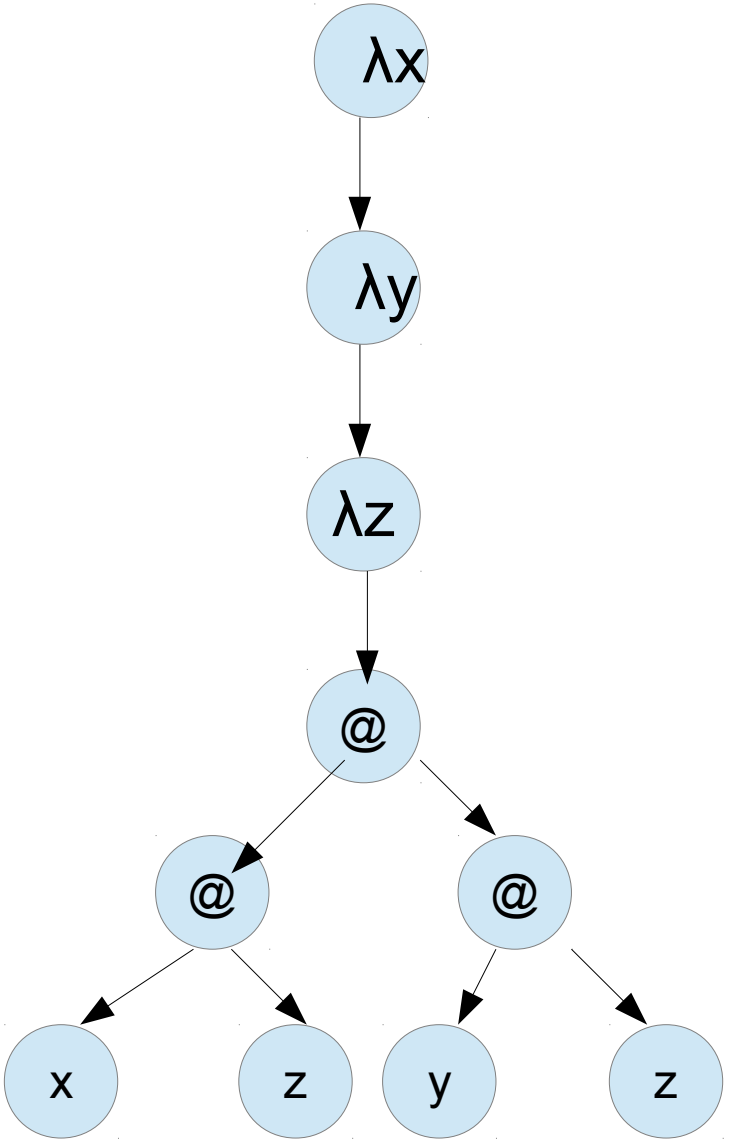
Příklady lambda termů

- λ x . x
- λ x y . x
- λ x y z . x z (y z)

Zase můžeme termy chápat jako stromy:





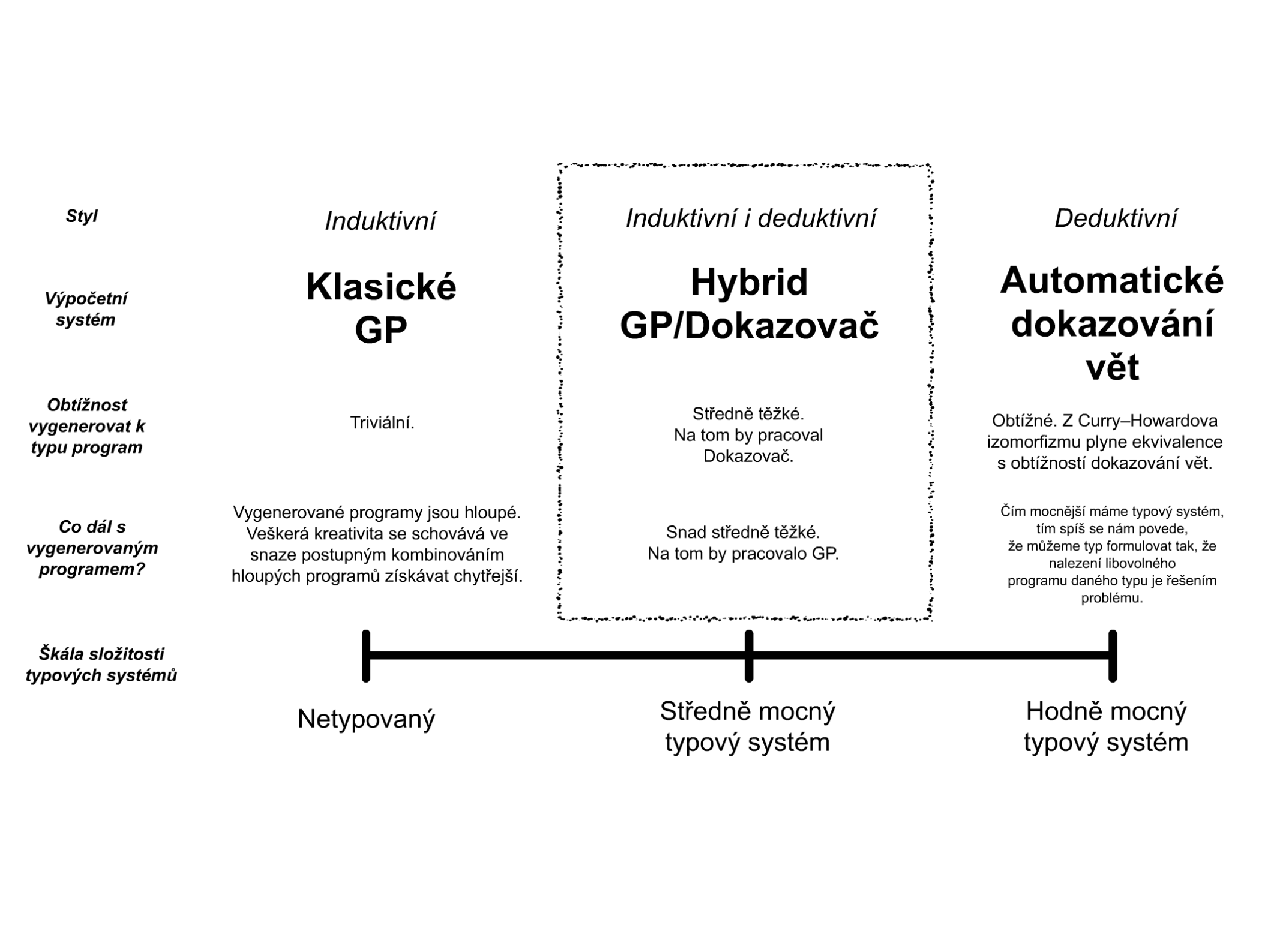


Program a jeho Typ

- Na program a jeho typ se dá koukat různě.
- Např těmito 3 způsoby:
  - Program $\in$ Typ
  - Typ je informace o Programu
  - Program je důkazem Typu jakožto tvrzení
    (Curry-Howardova korespondence)

Curry-Howardův izomorfismus

Svět logiky	Druh logiky	Formule	Důkaz
Svět typů	Typový systém	Typ	Program


Styl
Induktivní
Induktivní i deduktivní
Deduktivní
Výpočetní systém
Klasické GP
Hybrid GP/Dokazovač
Automatické dokazování vět
Obtížnost vygenerovat k typu program
Triviální.
Středně těžké.
Na tom by pracoval
Dokazovač.
Obtížné. Z Curry–Howardova izomorfizmu plyne ekvivalence s obtížností dokazování vět.
Co dál s vygenerovaným programem?
Vygenerované programy jsou hloupé.
Veškerá kreativita se schovává ve
snaze postupným kombinováním
hloupých programů získávat chytřejší.
Snad středně těžké.
Na tom by pracovalo GP.
Čím mocnější máme typový systém,
tím spíš se nám povede,
že můžeme typ formulovat tak, že
nalezení libovolného
programu daného typu je řešením
problému.
Škála složitosti typových systémů
Netypovaný
Středně mocný typový systém
Hodně mocný typový systém

Typ

- Obdobně jako pro lambda term můžeme induktivně definovat co je to typ:

  *(Nechť **α** je nějaký textový řetězec)*

- **α** je typ. (=atomický typ)
- Pokud **A** a **B** jsou typy, pak **(A→B)** je typ. (=typ funkce z A do B)

Γ⊢M:A

- Teorie typovaného λ kalkulu nám umožňuje odvozovat tvrzení následujícího tvaru:
- Γ⊢M:A
- Kde:
  - Γ je báze (nebo také kontext) : Mn. dvojic (proměnná,typ)
    - **Odopvídá náší množině stavebních prvků z GP.**
  - M je lambda term.
  - A je typ.

- Čteme to: Z báze Γ můžeme odvodit, že M je typu A.

3 odovozovací pravidla

$$[Axiom] \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$[E^{\rightarrow}] \qquad \frac{\Gamma \vdash M : A \rightarrow B, \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$[I^{\rightarrow}] \qquad \frac{\Gamma_x, x : A \vdash M : B}{\Gamma_x \vdash \lambda x.M : A \rightarrow B}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X}$$

$$\frac{\Gamma \vdash A \quad , \quad \Gamma \vdash A \to B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\frac{x \in \Gamma}{\Gamma \vdash x}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A \quad , \quad \Gamma \vdash A \to B}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash x : A \quad , \quad \Gamma \vdash f : A \to B}{\Gamma \vdash f\ x : B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x . M : A \to B}$$

Generování lambda termů

- Nám při generování termů jde o něco trochu jiného, než produkovat tvrzení tvaru $\Gamma \vdash M:A$.

- My chceme pro zadané Γ a A (tzn. mn. stavebních prvků a typ generovaného programu) vygenerovat takové M aby platilo:
  - $\Gamma \vdash M:A$

Z odvozovacích pravidel gramatiku

- Řekl jsem si, že by se hodilo vzít naše odvozovacích pravidla, a jen je přeformulovat na přepisovací pravidla „gramatiky“:

$$[Axiom] \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \approx \quad A_\Gamma \Longrightarrow x \qquad \text{kde } x : A \in \Gamma$$

$$[E^{\rightarrow}] \quad \frac{\Gamma \vdash M : A \rightarrow B, \Gamma \vdash N : A}{\Gamma \vdash MN : B} \quad \approx \quad B_\Gamma \Longrightarrow (A \rightarrow B)_\Gamma \; A_\Gamma$$

$$[I^{\rightarrow}] \quad \frac{\Gamma_x, x : A \vdash M : B}{\Gamma_x \vdash \lambda x.M : A \rightarrow B} \quad \approx \quad (A \rightarrow B)_\Gamma \Longrightarrow \lambda x.B_{\Gamma_x, x:A}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$A \rightarrow x$$

$$\frac{\Gamma \vdash x : A \quad , \quad \Gamma \vdash f : A \rightarrow B}{\Gamma \vdash f\ x : B}$$

$$B \rightarrow A \rightarrow B \ \ A$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x . M : A \rightarrow B}$$

$$A \rightarrow B \rightarrow \lambda\ x\ .\ B$$

!!!

Výhody a nevýhody tohoto generování

- **Výhoda:** Jednoduše se tam přidávají nové typové konstrukty (kartézský součin atd.), protože typové konstrukty mají k sobě definovaná odvozovací pravidla, která lze lehce převést na ta „gramatická“.
- **Nevýhoda:** Generované programy nemusí být v normální formě.

Trochu jinak

- V knize od Barendregt z roku 2010 [http://www.cs.ru.nl/~henk/book.pdf] je něco co se dá přepsat v té mé notaci jako:

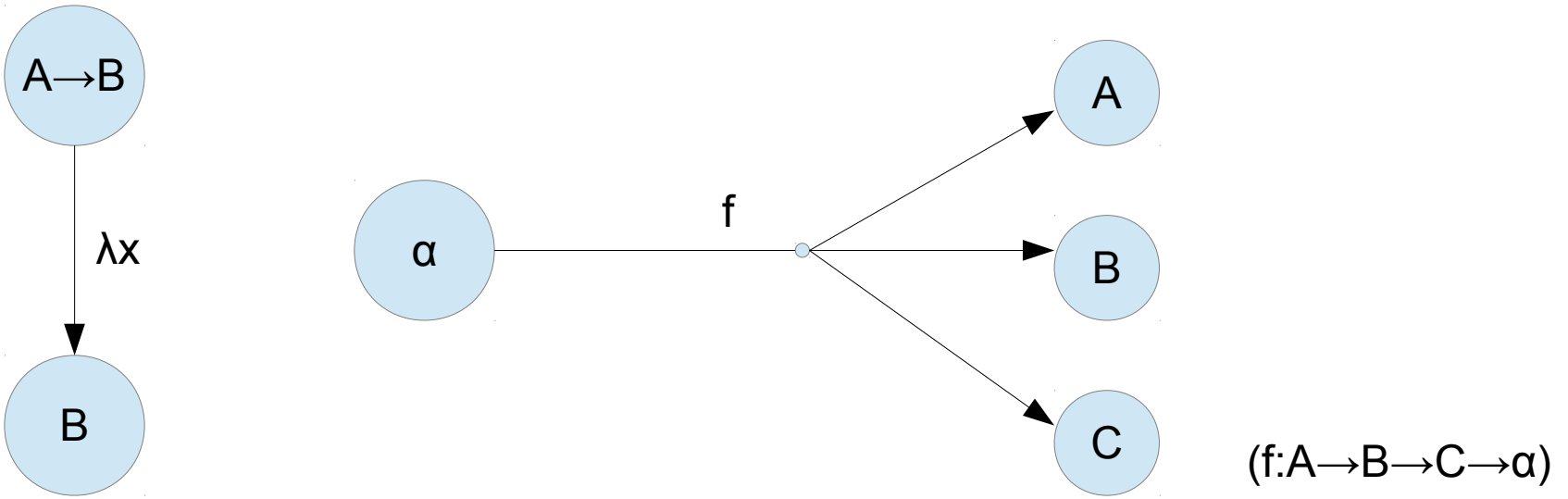
$$\alpha_\Gamma \Longrightarrow f\ (B_1)_\Gamma\ (B_2)_\Gamma\ \ldots\ (B_n)_\Gamma \qquad \text{kde } \alpha \text{ je atomický a } f : B_1 \to \ldots \to B_n \to A \in \Gamma$$

$$(A \to B)_\Gamma \Longrightarrow \lambda x.B_{\Gamma_x, x:A}$$

- A na základě těchto dvou pravidel následně popisuje koncept **Inhabitation Machine.**

Co je to Inhabitation Machine?

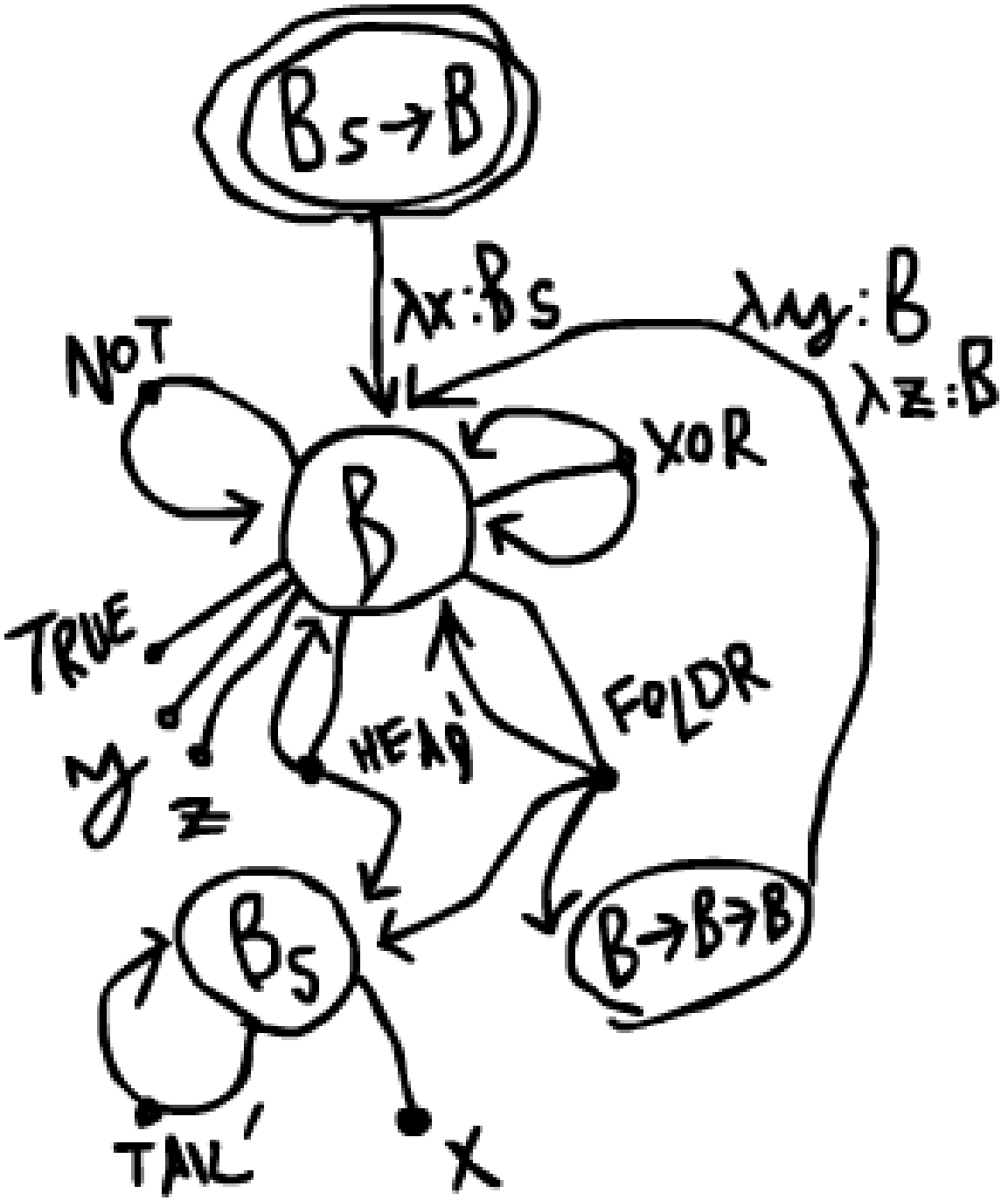
- Je to graf, který ma krom klasických hran ještě navíc „vidličkovité“ hrany. Neboli AND-OR graf.



- Vrcholy odpovídají typům, hrany odpovídají „kusům kódu“, které dáme na výstup.
- Začínáme ve vrcholu odpovídajícím typu generovaného programu.

Příklad IM

- Γ = {
  - foldr : (B→B→B)→B→Bs→B,
  - True:B,
  - not:B→B,
  - xor:B→B→B,
  - head':B→Bs→B,
  - tail': Bs → Bs }
- A = Bs → B


$B_S \to B$
$\lambda x : B_S$
$\lambda y : B$
$\lambda z : B$
NOT
XOR
B
TRUE
FOLDR
y
z
HEAD'
B_S
$B \to B \to B$
TAIL'
x

Jak křížit?

- Problém: Volné proměnné v uřezaných větvích!
- 2 řešení:
  - Vymyslet řešení opravující problémy s proměnnými
  - **Zbavit se proměnných**

SKI

- Libovolný term s proměnnými lze převést na term bez proměnných pomocí kombinátorů S,K a I.
  - S = λ x y z . x z (y z)
  - K = λ x y . x
  - I = λ x . x (Přičemž ale I = S K K)

    Fun fakt: typy S a K odpovídají logickým axiomům
    - K: $A \rightarrow (B \rightarrow A)$
    - S: $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Jak probíhá převod

- $T(x) = x$ ***(x proměnná)***
- $T(M\ N) = T(M)\ T(N)$
- $T(\lambda\ x\ .\ M) = K\ T(M)$ ***(x není v M volně)***
- $T(\lambda\ x\ .\ x) = I$
- $T(\lambda\ x\ y.\ M) = T(\lambda\ x\ .\ T(\lambda\ y\ .\ M)\ )$
- $T(\ \lambda\ x\ .\ (M\ N)\ ) = S\ T(\lambda\ x\ .\ M)\ T(\lambda\ x\ .\ N)$

Na co to jde použít?

- Díky tomu, že je to rozšíření/zobecnění GP tak na všechny předchozí aplikace, nyní však s možností pohodlně používat libovolnou sadu stavebních bloků.
- Dobrodružnější aplikace:
  - „Univerzální“ sada stavebních bloků
    - Můžeme se snažit o to, spojovat stavební sady pro různé problémy, tak aby GP stále našlo dobré řešení. Zlatý grál by pak byla univerzální sada stavebních bloků
  - **Šlechtění fitness** funkcí:
    - Ať už pro problém, kde je přirozená ff nevhodná.
    - Nebo při šlechtění „kritiků“ - např v computer generated art
  - Šlechtění samotných **střev GP** algoritmu.
    - Implementaci píšu v Haskellu a generované programy jsou také v Haskellu.
  - Metoda: **rozpadni známé řešení** a skus vymyslet lepší.
    - Máme-li například několik řešení různě relaxovaných problémů napsaných člověkem,mohli bychom tyto programy nějakým automatickým způsobem rozpadnout na stavební bloky a ty pak skusit šlechtit.