Typed Functional Genetic Programming

Tomáš Křen
tomkren@gmail.com

Jak funguje klasické *Genetické Programování*?

Fitness funkce představuje zadání řešeného problému.

Množina stavebních bloků dává součástky, ze kterých stavíme **Program řešící problém**.

Algoritmus generování počáteční populace ze stavebních bloků poskládá počáteční populaci - kandidáty na řešení problému.

A o zbytek se už postará algoritmus Genetického programování sám.

Chod GP je možno ovlivnit volbou vhodných **parametrů**.

My půjdeme ve šlépějích **Johna Kozy** (vynálezce GP) a záměrně se nebudeme snažit o optimalizaci řešení chytrou volbou těchto **méně důležitých parametrů**.

Tím se vyhneme *„šamanismu s konstantami“*.

Algoritmus Genetického programování

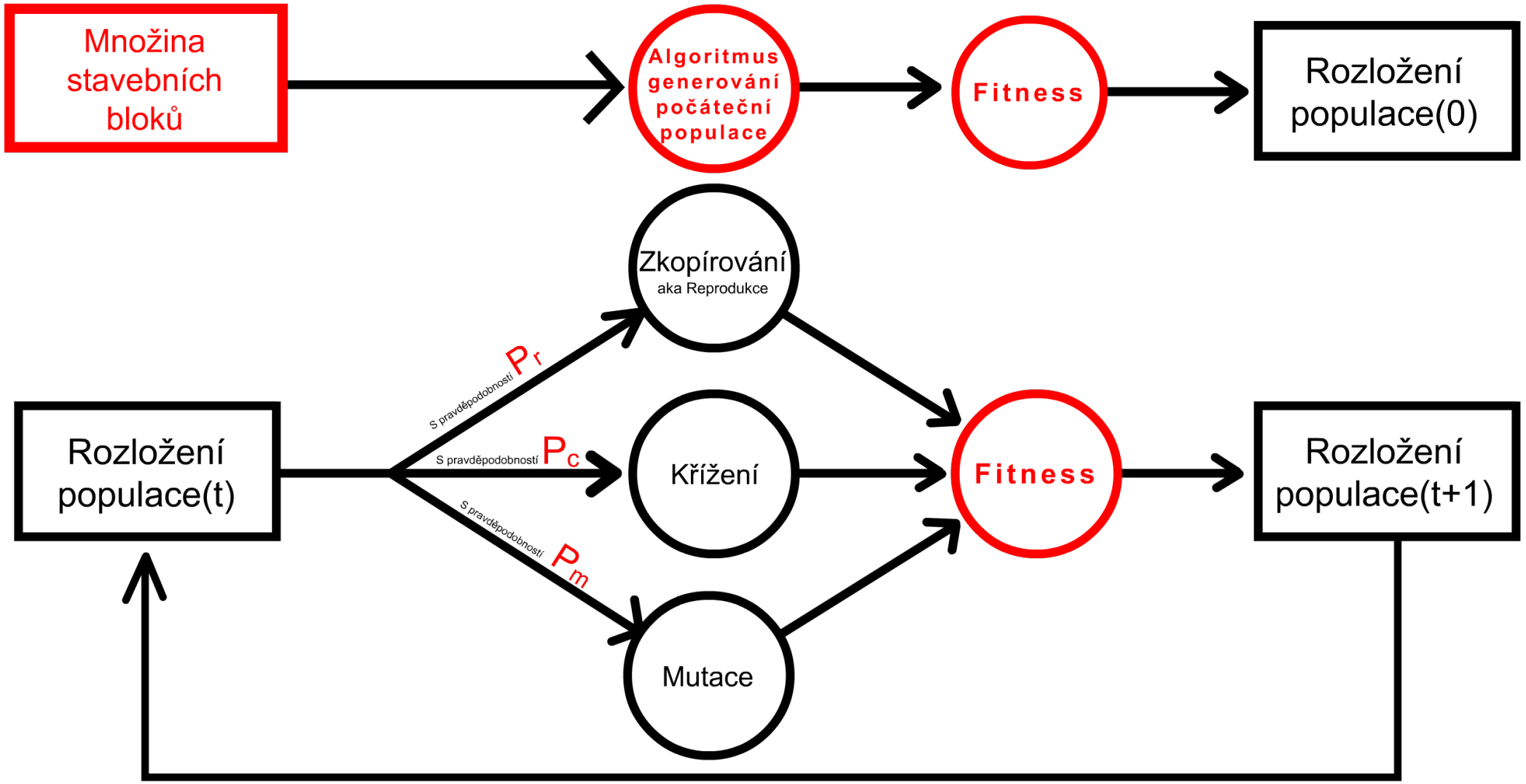
Červeně jsou označeny vstupy GP algoritmu.

V prvním kroku **Algoritmus generování počáteční populace** sestaví ze **stavebních bloků** populaci v čase **0**.

Počáteční populaci ohodnotíme pomocí **finess**, což nám dává „**pravděpodobnostní rozložení“ populace** v čase **0**.

Krok algoritmu v čase **t** odpovídá transformaci **rozložení populace** v čase **t** na **rozložení populace** v čase **t+1**.

Z populačního rozložení se vybere jedin(ec/ci) a provede se na něm jedna z transformací (**reprodukce** (tzn. identita) , **křížení** nebo **mutace**).



Proč typovaně a čistě funkcionálně?

Typovanost

- **Klasicky** se stavějí **Lispové *s-výrazy*** (tzn. stromy). Stavebními bloky jsou funkce (listy jsou konstanty, kteréžto můžeme chápat jako nulární funkce).
- **Problémem** je funkce propojit tak, aby jejich propojení dávalo **smysluplný program.**
- Klasicky se to řeší tak, že funkce operují s **jediným typem dat** – takže libovolné propojení je (syntakticky) smysluplné.
- Pro zvládnutí složitějších situace se nabízí **využít typy**. Povolíme stavbu jen takových programů, které neporušují typová pravidla.
- Zaváděním vhodných typů a tím, že řekneme „řešení je typu T“ **popisujeme** velice **deklarativním způsobem** charakter **řešení**. (Můžeme vidět jakousi posloupnost podle míry „deklarativnosti“: Procedurální programování → Deklarativní programování → Typové programování.)

Čistě funkcionálně

(generovaná řešení i samotná implementace)

- Se stavem jsou jenom potíže – **zesložiťuje možnost přemýšlení o kódu**. Přesněji to máme ověřeno u přemýšlení o kódu napsaném člověkem. Máme důvodné podezdření myslet si něco podobného i o kódu napsaném strojem.
- Zkušenosti z funkcionálního programování ukazují, že i **minimalistickými syntaktickými prostředky** je možno úsporně popsat **složité koncepty**.
- Pokud si dáme tu práci a naimplementujeme systém funkcionálně, tak máme mnohem širší pole působnosti při nějakých **„meta-nápadech“**.

Od množiny bloků ke gramatice

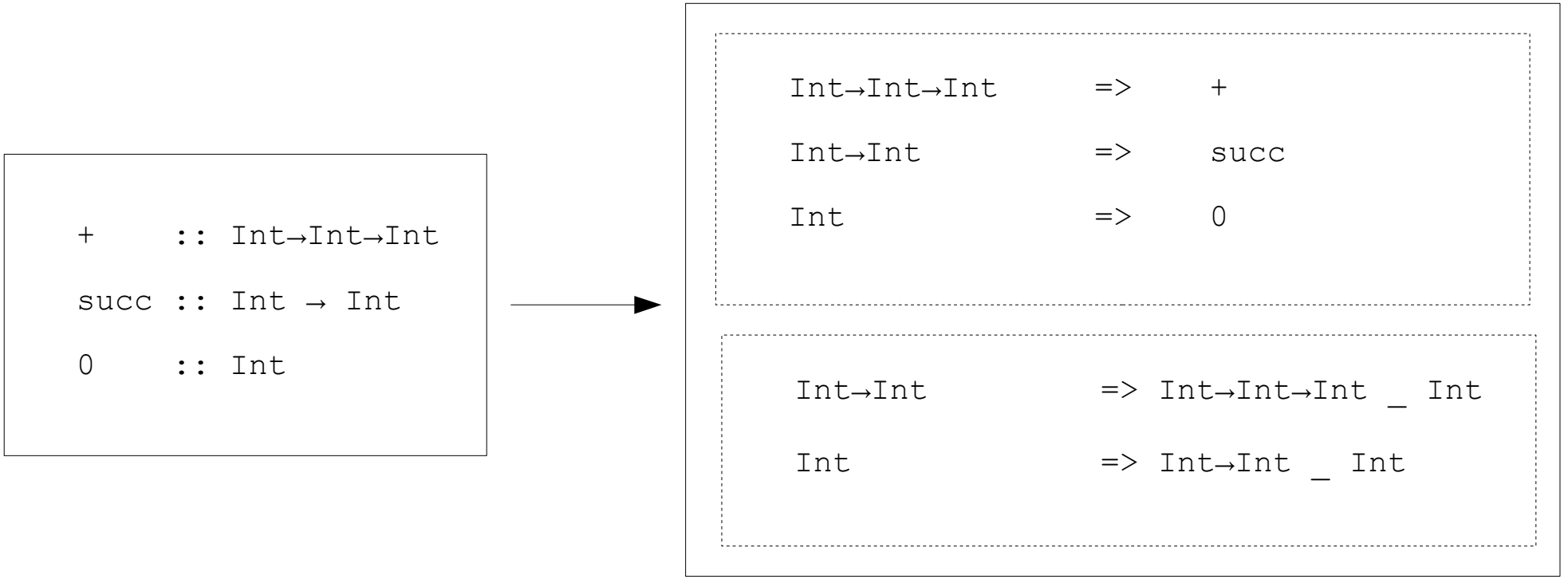
Naší **množinu stavebních bloků** tvoří **otypované funkce**. Nám by se při tvoření programů **hodila gramatika**. Ukážeme, jak je možné transformovat takovéto stavební bloky na bezkontextovou gramatiku.

Terminály gramatiky jsou funkční symboly (a _ jakožto aplikace funkce).

Neterminály gramatiky jsou typy.

Pro každou funkci *f* typu *t* dostváme pravidlo : `t => f`

Pro každý typ funkce (nějaké funkce z množiny) tvaru (*a→b*) pravidlo : `b => a→b _ a`

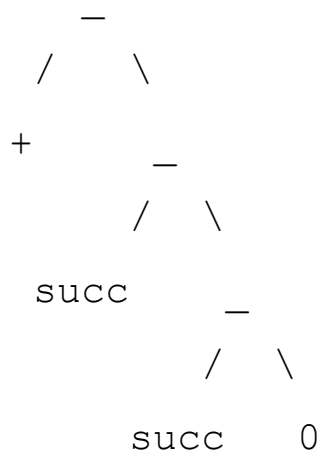


** Pro lepší názornost vynecháváme v pravidlech závorky, které by zajistily dobré uzávorkování vzniklých výrazů.*

Příklad

```
Int→Int                          | Int→Int      => Int→Int→Int _ Int
Int→Int→Int _ Int                | Int→Int→Int  =>  +
+ _ Int                          | Int          => Int→Int _ Int
+ _ (Int→Int _ Int)              | Int→Int      => succ
+ _ (succ _ Int)                 | Int          => Int→Int _ Int
+ _ (succ _ (Int→Int _ Int))     | Int→Int      => succ
+ _ (succ _ ( succ _ Int))       | Int          => 0
+ _ (succ _ ( succ _ 0 ))
```

Což je funkce, která k číslu přičte 2ku



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

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

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

Curry-Howardův izomorfismus

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

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

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

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

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

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

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

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

$$A \rightarrow x$$

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

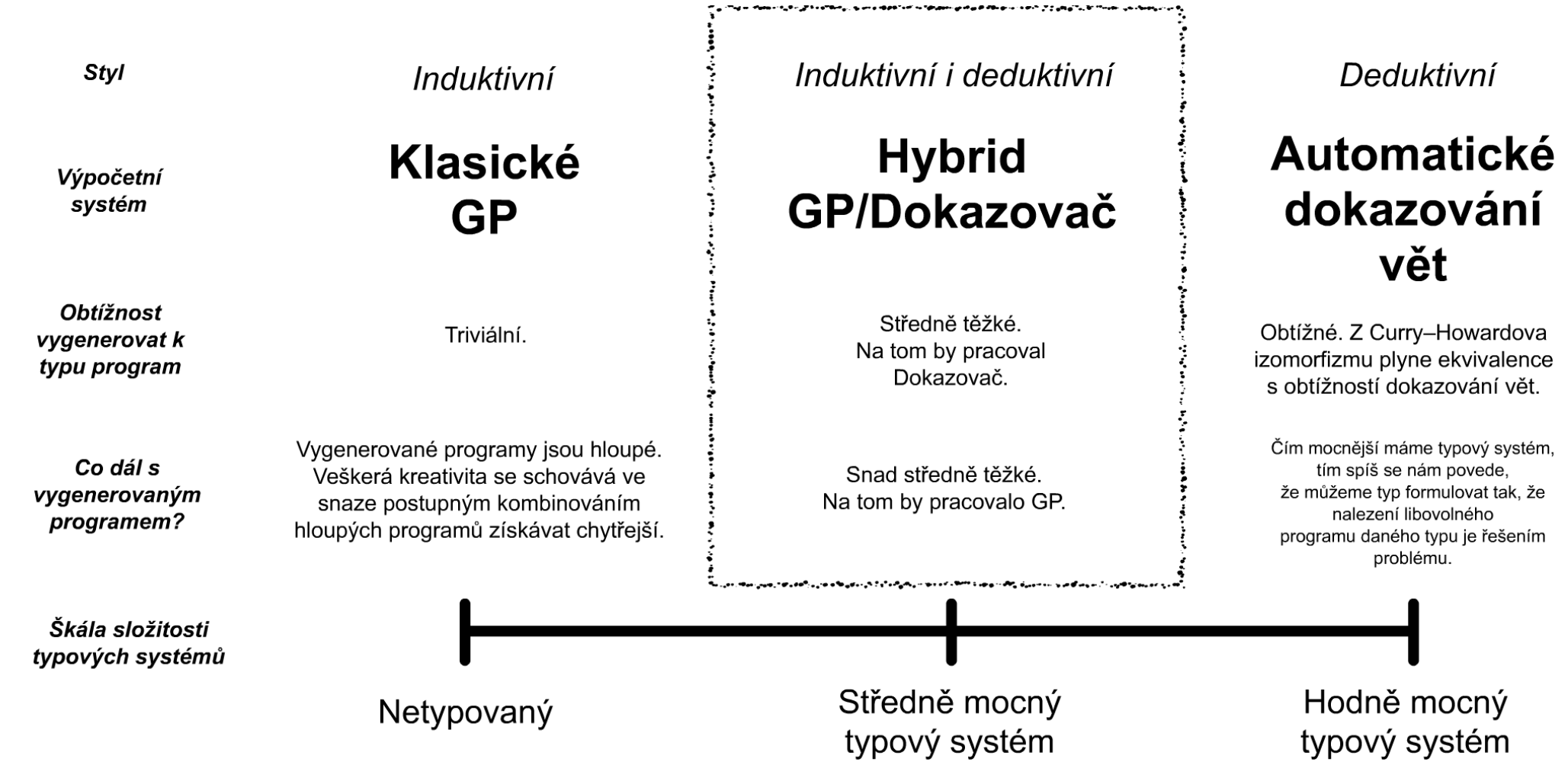
$$B \rightarrow A \rightarrow B \ \ A$$

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

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

!!!

Odhad vlastností „Typovaného GP“ v závislosti na složitosti použitého typového systému



Jak naprogramovat programátora?

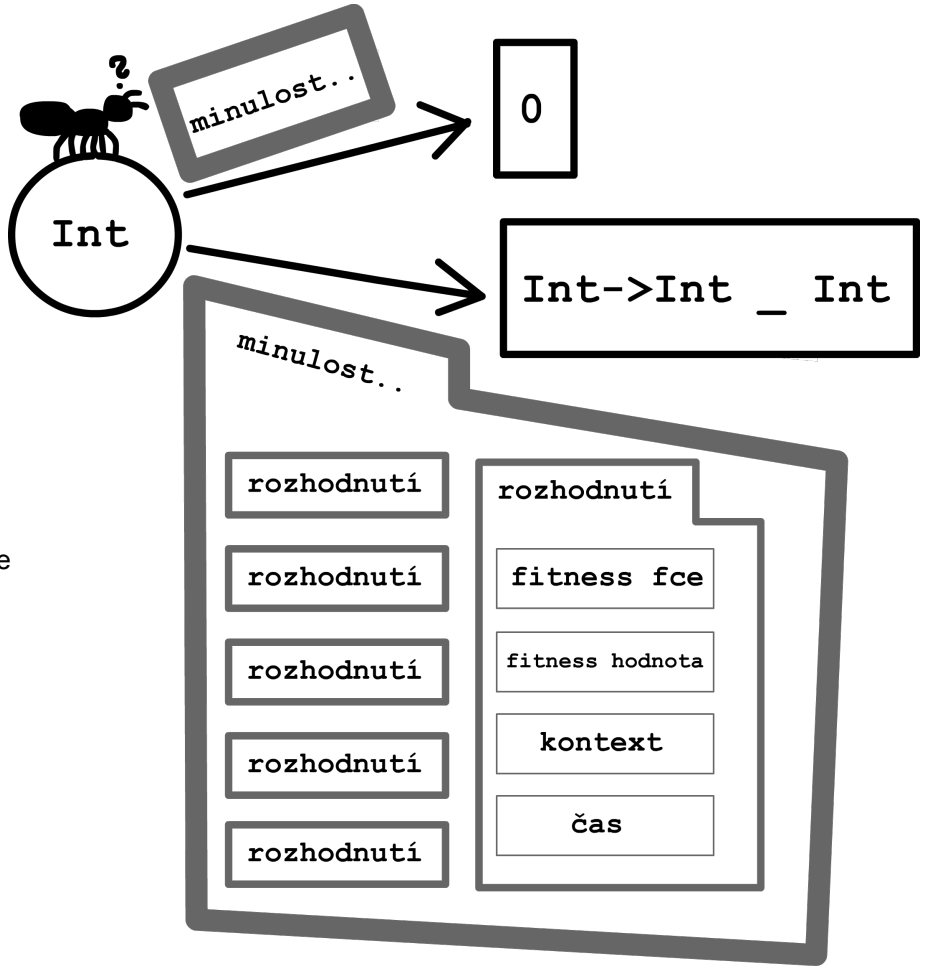
- Pokusme se neformálně podchytit pojem programátora.
- Programátor je systém který **pro zadaný problém napíše program** řešící takovýto problém.
- Programátor při řešení nového problému **využívá zkušenosti** nabyté při řešení předchozích problémů.
- Abychom mohli považovat systém založený na GP za programátora, hodilo by se **omezit nutnost zadávat** na vstupu jakékoli **jiné parametry než fitness funkci** (= zadání problému).
- **Tzn** chceme se **zbavit nutnosti specifikovat** při řešení problému specifickou **množinu stavebních bloků** a specifický **algoritmus generování** počáteční populace.
- Místo toho aby měl programátor pro každý problém speciální sadu stavebních funkcí, bude mít **jednu univerzální sadu stavebních funkcí**. Na tuto sadu se můžeme koukat jako na jeho **programovací jazyk**. (Na ten je celkem rozumné se v tuto chvíli dívat jako na společný všem programátorům.)
- **Algoritmus generování** počáteční populace bude **součástí programátora**. To v čem se liší dva různí programátoři je právě tento algoritmus.

Termiti aneb *„evoluce proteinů“*

- Pro popis toho, jak by mohl být řešen algoritmus generování počáteční populace v jednotlivých programátorech, si zavedeme pojem *termit*.
- Termitem rozumíme **program schopný generovat programy podle gramatiky**.
- **Jeden krok** činnosti termita spočívá v **transformaci řetězce gramatiky** na jiný řetězec gramatiky použitím nějakého pravidla gramatiky.
- Programátor si udržuje nějaké **pravděpodobnostní rozložení termitů**. Ve chvíli, kdy je potřeba vygenerovat jedince počáteční populace, vybere z tohoto rozložení nějakého termita a nechá ho postavit požadovaného jedince.
- Na toto **rozložení** se můžeme koukat **jako** na **genetickou informaci programátora**. Pokud půjdeme v této analogii dál, dostaneme, že jeden termit odpovídá jednomu genu, tedy zakódování jednoho proteinu.
- Pokud bychom přistoupili na variantu, že toto rozložení se **vyvíjí společně s programátorem**, tak se nabízí koukat na jednoho termita místo jako na *gen* jako na ***mem***.
- Pro realizaci takovéhoto dynamického rozložení se nabízí použít **zase algoritmus GP**. Mám na mysli následující:
  - Vyřešení jednoho problému, který zadáme programátorovi vyžaduje jeden **běh** GP algoritmu.
  - Tento celý běh pak poslouží jako jeden **krok** (případně jako jeden *fitness case*) v GP evolvujícím **termity** daného programátora.

Termiti a ACO

- Máme nepřeberně mnoho možností, jaké termity použít.
- Např. můžeme vytvořit termita chovajícího se po vzoru **klasického Kozova algoritmu** pro generování počáteční populace. To ukazuje na fakt, že takto rozšířené GP je nadmnožinou toho klasického.
- Podívejme se nyní na jedno z možných pojetí termitů, inspirované *Ant colony optimization*.
- Představme si, že každý termit má navíc k dispozici **„studnici vědomostí“**, která obsahuje informace **o předchozích rozhodnutích termitů** (rozhodnutím máme na mysli volbu přepisovacího pravidla pro nějaký netrminál gramatiky). Do této studnice termit nahlédne ve chvíli, kdy si vybral nějaký neterminál k přepsání a váhá které pravidlo si má vybrat.
- Studnice pro požadované přepisovací pravidlo vydá seznam minulých rozhodnutí, u každého rozhodnutí jsou tyto informace:
  - **Jaký problém** byl řešen, když k tomuto rozhodnutí došlo. (tzn. kód fitness funkce)
  - **Jaká** byla **hodnota fitness** funkce pro vyprodukovaný program, na jehož produkci se podílelo toto rozhodnutí.
  - **Jak vypadal řetězec gramatiky**, než došlo k aplikaci přepisovacího pravidla. (to nazvem kontext)
  - **Čas** kdy došlo k tomuto rozhodnutí.
- Termit si pak na základě těchto informací porovnáním se svou současnou situací vybere nějaké přepisovací pravidlo a to aplikuje. Chytré by bylo, kdyby uvažoval následovně:
  - Chci, aby tenkrát řešený problém byl **podobný jako můj současný problém**.
  - Chci, aby **fitness hodnota** tenkrát byla **co největší**.
  - Chci, aby **kontext** tenkrát a teď byly **co nejpodobnější**.
  - Chci, aby **čas** byl **co nejblíž teď.** (Inspirováno vypařováním feromonů u mravenců.)
- Pro efektivní vykonávání výše naznačených úvah by se termitovi hodilo mít k dispozici dva užitečné typy funkcí:
  - **Metrika na kódech fitness funkcí**
  - **Metrika na kontextech**



ATP aneb měna krytá *β-redukcemi*

- *Úvaha na úvod:* V současné chvíli můžeme **proces programování** přirovnat k **centrálně plánované ekonomice**. Společenskou hierarchii tvoří strom programu (tzn. volající funkce je ve společensky vyšší roli než volaná funkce). Programátor (nyní máme na mysli člověka :)) je úplně navrchu (spouští program). Řízením ekonomiky zde chápeme řízení výpočetní složitosti. Za tu má plnou odpovědnost programátor. Funkce slepě poslouchají centrální plán.

- Můžeme se pokusit **experimentovat s tržností zavedením kreditů/peněz/energetické jednotky** – nazvěme tuto jednotku **ATP**. Jedna možnost je např. takto: Společně s voláním funkce musí volající navíc specifikovat kolik ATP dává k dispozici pro výpočet. Tento obnos se mu odečte ze současného konta. Volaná funkce pak má k dispozici zadané množství ATP. Zahájí svůj výpočet a pokud se jí ho povede dokončit (tzn. vystačí s ATP) vrátí výsledek a zbylé ATP. Pokud to nestihne, vrátí *nedopočítaný výpočet* (a 0 ATP). Volající funkce pak pokračuje v zavislosti na svém současném kontě dál...

- Důležité je, aby byl systém navržen tak, že každá elementární operace ubere jedno ATP. **Ideálně 1 ATP na 1 β-redukci.** (Přesný mechanizmus zatím nemám rozmyšlený.)

- Toto podle mě dost souvisí **s redukčními strategiemi**. Myslím, že by se to dalo považovat za svého druhu redukční strategii, případně by bylo zajímavé promyslet souvislost/souhru s *lazy* strategií. (Ale toto také nemám dostatečně promyšleno.)

- Výhodou je, že nyní nemusíme počítat výpočetní složitost, ale „určujeme ji“ (ovšem tuto krásnou ideu hatí to, že program samozřejmě nemusí doběhnout). To má evidentní výhody v systému, kde programy vznikají náhodně.

- Nevýhodou je s ATP spojená výpočetní režie.

Trh problémů

- Nyní se pokusíme nastínit náčrt high-level pohledu na celý systém.
- Jedná se o **multiagentní systém**, kde jednotlivý agenti jsou dříve popsaní **programátoři**. Nebo přesněji řečeno jejich ústřední výpočetní jednotkou je programátor (chceme, aby programátor byla funkce jako každá jiná, kdežto implementace agenta by si mohla vyžádat nějaké vyšší nároky, plus akce spojené s koordinací života v tomto systému nejspíš nebudou úplně programátorského rázu).
- **Tito agenti mají konto s ATP** (je zde drobný ale podstatný rozdíl od funkcí, které mají své lokální konto dáné voláním). Pro svou činost potřebují toto ATP.
- Komunikace s agenty probíhá formou **„centrální nástěnky“**. Uživatel na tuto nástěnku umístí problém, rozpočet, případnou odměnu, odhadovanou obtížnost či jiné informace upřesňující charakter „nabídky na práci“.
- Agenti si v rámci své činnosti **naplánují** své další **vzbuzení** do „kalendáře“. Krok systému spočívá ve výběru prvního agenta z kalendáře. Ten pak provede akce odpovídající jeho programu.
- **Součástí akce**, kterou se agent rozhodne provést, může být **pokus o splnění** nějaké **„nabídky na práci“**.
- Nástěnka může být využita i **k obchodování mezi agenty samotnými,** například pro „obchod s termity“. Nebo dokonce pro vytváření nabídek na práci samotnými agenty.
- Určitě by bylo zajímavé klást do systému koumácké dotazy na systém samotný a tak **zvyšovat jeho povědomí sama sebe.**
- Samozřejmě takovýto systém je **vzdušný zámek** vybudovaný na předchozích úvahách, ale myslím že je zdravé přiznat, kterým směrem moje snažení směřuje.

Literatura

- *Koza, J.R. (1992).* **Genetic Programming: On the Programming of Computers by Means of Natural Selection**, MIT Press. ISBN 0-262-11170-5
  [Četl jsem, pro mě bible (nebo minimálně Genesis) GP.]

- *Koza, J.R., Keane, M., Streeter, M., Mydlowec, W.,Yu, J., Lanza, G. (2005).* **Genetic Programming IV: Routine Human-Competitive Machine Intelligence**, Springer. ISBN 978-0-387-26417-2
  [Mám rozečtený začátek, nejnovější z Kozovy série – mimojiné je zde popisován produkt GP který má svůj vlastní patent.]

- *Poli, R., Langdon, W. B., McPhee, N. F. (2008).* **A Field Guide to Genetic Programming.**
  Lulu.com, freely available from the internet. ISBN 978-1-4092-0073-4.
  [Informačně velice nahuštěné. Kvalitní přehled a spousta referencí.]

- *Yu, T. (2001).* **Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction**
  [Článek o modelování rekurze pomoci fold a dalších věcech, teď čtu.]

- *Montana, D. (1994).* **Strongly Typed Genetic Programming**
  [Další článek o GP v Haskellu, zatím jsem nečetl.]

- *Jones, S. P. (1991).* **Implementing Functional Languages: a tutorial**
  [Výborná učebnice o překladačích funkcionálních jazyků, mám rozečteno. Je myslím potřeba zvláště pro solidní zvládnutí věcí okolo ATP.]

- *Wu, C. (2005).* **A multi-agent framework for distributed theorem proving**
  [Článek o dokazování vět v multiagentním prostředí, zatím jsem nečetl.]

- *Paulson, L. (1985).* **Natural Deduction as Higher-Order Resolution**
  [Článek o dokazovači vět v intuicionistické logice pomocí higher-order resoluce. Četl jsem kus.]