Typed Functional Genetic Programming

Tomáš Křen

V tomto povídání bych chtěl neformálně popsat dosavadní vývoj a současný stav práce *Typed Functional Genetic Programming*.

Cílem mého aktuálního snažení bylo udělat v jazyku Haskell první prototyp systému řešící úlohu genetického programování nad typovaným funkcionálním jazykem. Zatím jsem nekladl příliš důraz na robustnost a také jsem se omezil na minimum víceméně triviálních problémů, na kterých systém testuji. Přijde mi totiž, že cílem prvního prototypu je hlavně ujistit se, že plánovaná architektura systému je realizovatelná; zjistit jaká skrytá úskalí problém skýtá; upřesnit představu o tom, na jaké podproblémy se problém dělí a načerpání nových nápadů.

Podívejme se v rychlosti na to, co to je GP-úloha a jak jí GP-systém řeší. V nejhrubším přiblížení je zadaní GP-úlohy *fitness funkce* – zobrazení z prostoru programů do nezáporných reálných čísel. Řešení GP-úlohy je program. Fitness funkce má jednoduchou interpretaci: Čím vyšší číslo dává programu, tím je tento program lepším řešením.

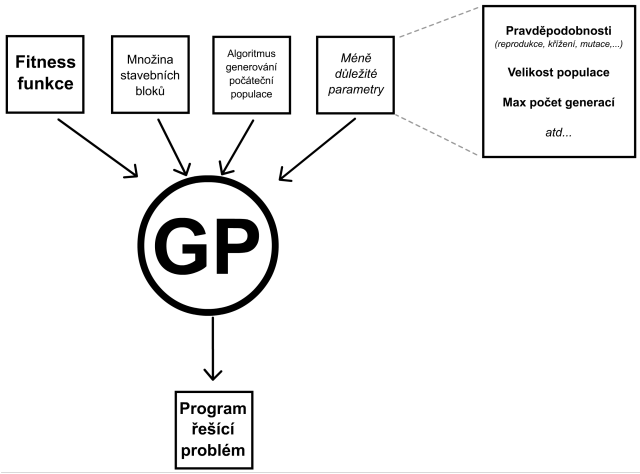
Při o něco jemnějším přiblížení za zadání GP-úlohy můžeme považovat následující čtveřici (pětici): Fitness funkce, (typ řešení), sada stavebních bloků, počet *generací* a počet *jedinců*.

Fitness funkce (FF) je funkce typu *"typ_řešení* $\rightarrow \mathbb{R}_0^+$".

Sada stavebních bloků neboli *enviroment*: Řešení GP-úlohy je program. Program se typicky skládá z konstruktů jazyka, proměnných a *symbolů vestavěných či jinde definovaných funkcí (případně konstant)*. Sadou stavebních bloků máme na mysli ty symboly funkcí/konstant, které se mohou vyskytovat v řešení (společně s jejich implementacemi a typy).

GP-systém si nejdříve vygeneruje určitý počet programů (*kombinátorů*) – programy představují "jedince". Tito na začátku vygenerovaní jedinci představují první "generaci".

GP-systém (inspirován přirozeným výběrem v přírodě) pak na základě n-té generace a FF vytvoří (n+1)-ní generaci jedinců (jak to dělá, zatím ponechme stranou). Za celkové řešení je pak prohlášen nejlepší jedinec ze všech generací.



Zaměřme nyní pozornost na první fázi GP-výpočtu – generování první generace jedinců. Zadaní GP-úlohy říká jakého typu je jedinec – tedy zde máme úlohu hledaní termů obydlujících

zadaný typ. Vzhledem ke Curry–Howardově korespondenci se můžeme na tento problém dívat jako na úlohu hledaní důkazů zadané formule.

Zde jsem stál před rozhodnutím, zda použít nějaký již hotový intucionistický dokazovač a nebo zda se pokusit tento problém řešit sám. Nakonec jsem se rozhodl udělat si "dokazovač" sám a to hlavně z následujícího důvodu: Situace při dokazování logických formulí je taková (tedy přesněji já jí tak laicky odhaduju), že chceme dokázat co možná nejsložitější formuli co možná nejkratším důkazem, přičemž nám nejde až tolik o to, že bychom chtěli co možná nejvíc různých důkazů. Naproti tomu naše situace je taková, že dokazujeme poměrně triviální formule, ale "důkazů" chceme co možná nejvíce a pokud možno rozmanitých co do délky.

Myšlenka metody, kterou jsem zvolil je založená na tom, že budeme pracovat s "rozpracovanými" lambda termy. Mimo proměnné/konstanty, aplikace a lambda abstrakce ještě navíc budeme uvažovat term sestávající z *typového termu* a *báze* – říkejme mu *neterminál*. Interpretace neterminálu je takováto: Neterminál je "rozepsaný podterm", u kterého víme zatím jen jeho typ a jaké symboly (a k těmto symbolům jejich typy) můžeme při jeho pozdějším "dodělání" použít. Dovolil jsem si pro netrminál typu σ s bází Γ vymyslet značku σ_Γ. Takže například $\left(\lambda\left(x:\sigma\right).\tau_{\{x:\sigma,f:\sigma\to\tau,g:\sigma\to\sigma\}}\right)$je rozdělaný term pro funkci typu $\sigma\to\tau$ s již "hotovou hlavou" (proměnná x) ale ještě "nehotovým tělem", o kterém však už víme, že je typu τ a uvnitř něhož smíme používat symboly x,f a g.

Výhoda takto rozšířené definice je, že nám umožňuje pro odvozovací pravidla (která zadávají náš typový systém) celkem přímočarou metodou najít jim ekvivalentní "redukční pravidla" nad rozdělanými termy. Každou redukci pak chápeme jako jeden krok důkazu.

Pro tři nejoblíbenější odvozovací pravidla to vypadá následovně:

$$[Axiom] \qquad \frac{x:\sigma\in\Gamma}{\Gamma\vdash x:\sigma} \qquad \approx \qquad \sigma_\Gamma \Longrightarrow x \qquad \text{kde } x:\sigma\in\Gamma$$

$$[E^{\to}] \qquad \frac{\Gamma\vdash M:\sigma\to\tau,\Gamma\vdash N:\sigma}{\Gamma\vdash MN:\tau} \qquad \approx \qquad \tau_\Gamma \Longrightarrow (\sigma\to\tau)_\Gamma\ \sigma_\Gamma$$

$$[I^{\to}] \qquad \frac{\Gamma_x,x:\sigma\vdash M:\tau}{\Gamma_x\vdash\lambda x.M:\sigma\to\tau} \qquad \approx \qquad (\sigma\to\tau)_\Gamma \Longrightarrow \lambda x.\tau_{\Gamma_x,x:\sigma}$$

Pro odvozovací pravidla pro součiny a součty lze postupovat obdobně (poznamenejme, že v současném prototypu je zatím jen Simply typed lambda calculus).

Napsal jsem "redukční pravidla" v uvozovkách, protože fáze vyhodnocování (tzn. beta redukce) a generování (tzn. tato "redukční pravidla" odpovídající krokům důkazu) jsou od sebe v současném prototypu oddělené, a tak se takovýto pohled může zdát zavádějící. V budoucnu by se to ale mohlo změnit. Navíc mi přijde, že se pak o tom hezky přemýšlí (ale možná je tahle intuice scestná): klasicky se koukáme na redukci jako na něco co posouvá nehotový výpočet k hotovějšímu - náš rozšířený pohled umožňuje nejenom nehotovost co se týče míry "spočítanosti výpočtu" ale také co se týče míry "zadanosti výpočtu".

Budu preferovat stručnost před přesností a nebudu zde moc popisovat technickou realizaci dokazovače. V zkratce je to realizováno následovně: Takováto "redukční pravidla" zanášejí do celé věci mohutný nedeterminizmus - v tom smyslu že neterminály se mohou nakonec redukovat na velké množství různých hotových termů. Tedy se musí prohledávat. Zatím jsem implementoval dva příbuzné dokazovače. První popíšu teď, druhý až o něco dále v textu.

K prohledávání používáme *A* algoritmus*. Prohledáváme prostor našich termů, přičemž hledáme termy s nulovým počtem neterminálů. Minimalizujeme počet kroků důkazu. Pro

A* potřebujeme heuristiku, která by (zespoda) odhadovala potřebný počet kroků k dokončení důkazu. Jako tento odhad používám počet neterminálů (odhaduje zespoda protože minimálně se musíme zbavit každého terminálu což zabere minimálně jeden krok) - a ač je to celkem přímočarý/hloupý odhad, tak mě překvapilo jak rychle to dokazuje (ale přiznávám, že sem to nejspíš nezkoušel na ničem moc komplikovaném). Prohledávání zahajujeme z termu "$typ\_řešení_{enviroment}$".

Ještě než se dostaneme k druhé části GP-algoritmu, bych se chtěl zmínit o myslím praktické datové struktuře Dist, kterou jsem pro účely práce navrhnul.

Dist představuje implementaci distribuce (pravděpodobnostního rozložení). Distribucí máme na mysli datovou strukturu, obsahující prvky a k nim asociované pravděpodobnosti.

Typ této datové struktury je parametrizován typem prvků distribuce.

Nad touto strukturou máme tři základní operace:

- Vytvoření distribuce .... O(n)
- Náhodný výběr z distribuce .... O(log n)
- Náhodný výběr z distribuce s odebráním vybraného prvku .... O(log n)

Vytvořit distribuci můžeme dvěma způsoby:

Jednodušší způsob je pomocí seznamu hodnot a k nim přiřazených "pravděpodobností". Přesněji řečeno, se nejedená o pravděpodobnosti ve formálním smyslu a to v tom, že jejich součet nemusí být 1, Dist už se sám postará o normalizaci. Navíc se informace o původních hodnotách nezapomíná, což je mnohdy příjemná vlastnost. Např. když chceme populaci jedinců reprezentovat jako distribuci jedinců, kde pravděpodobnosti odpovídá jejich fitness. Tímto přístupem můžeme uchovat obě informace (tzn. fitness i pravděpodobnost) úsporně a na jediném místě.

Příklad: `[('a',5),('b',35),('c',10)]` bude odpovídat pravděpodobnostnímu rozložení písmen, přičemž pravděpodobnost `'a'` je 0.1, `'b'` je 0.7, `'c'` je 0.2 a všech ostatních písmen 0.

Složitější způsob zadání distribuce umožňuje mimo seznamu s prvky `(`*hodnota*`,`*pravděpodobnost*`)` i prvky `(`*"funkce distribuce"*`,`*pravděpodobnost*`)`. Funkcí distribuce zde máme na mysli funkci z intervalu $< 0, 1 >$do *"hodnot"*. [TODO zjistit jak se "funkce distribuce" jmenuje oficiálně] Takto zadaná distribuce má přímočarou interpretaci: z rovnoměrného rozdělení na intervalu $< 0, 1 >$vybereme náhodné číslo, na nějž aplikujeme danou funkci, čímž dostaneme vybraný prvek. (Poznamenejme, že první způsob zadaní by šlo úplně vypustit, a simulovat ho konstantní "funkcí distribuce" což ovšem z praktických důvodů nebudeme dělat.)

Datový typ Dist nám tímto způsobem umožňuje pohodlně skládat jednodušší distribuce do složitějších.

Příklad: `[ ( (*2) , 40 ) , ( (+10) , 60 ) ]` odpovídá rozložení, kde s pravděpodobností 0.4 bude vybráno číslo z intervalu $< 0, 2 >$ (každé z nich se stejnou pravděpodobností) a s pravděpodobností 0.6 bude vybráno číslo z $< 10, 11 >$ (každé z nich se stejnou pravděpodobností).

Dist je vnitřně reprezentován binárním stromem. Z důvodu efektivnějších operací si dále uchovává informaci o počtu listů a součtu (nenormalizovaných) pravděpodobností. Listy bin. stromu jsou dvou typů, odpovídající dvěma typům konstrukčních dvojic *(hodnota,pst)* respektive *(distr. fce, pst)*. Nelistové uzly bin. stromu obsahují "dělící" hodnotu z intervalu $< 0, 1 >$.

Tato dělící hodnota je rovna součtu normalizovaných pravděpodobností prvků v levém podstromu.

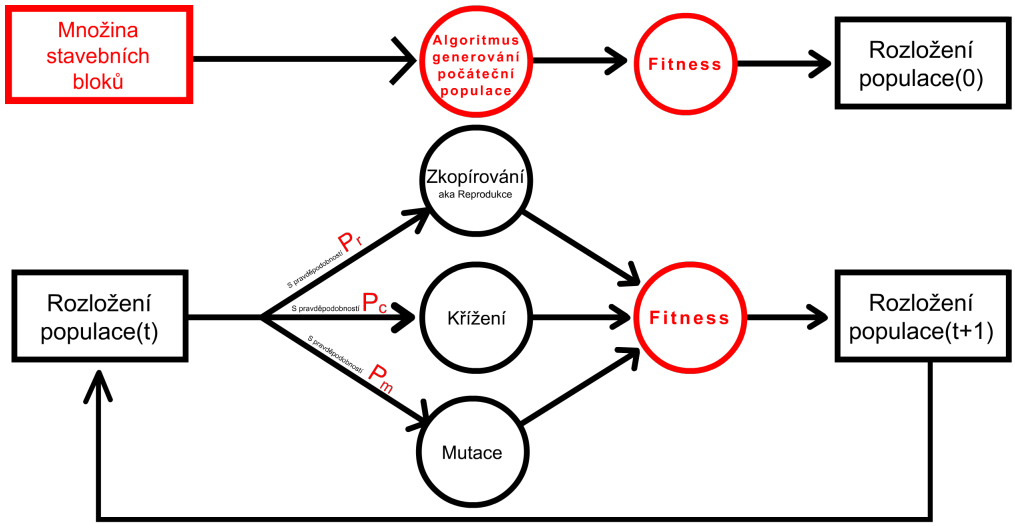
Neformálně funguje výběr z distribuce následovně: Náhodně vybereme číslo z intervalu <0,1>, kteréžto pošleme kořeni stromu. Pokud je toto číslo menší než dělící hodnota tohoto uzlu, vybíraný prvek se nachází v levém podstromu; jinak se nachází v pravém podstromu. Toto číslo vhodným způsobem přepočítáme a odešleme zvolenému podstromu, kde se postup rekurzivně opakuje. Pokud narazíme na list prvního typu, vracíme v něm obsazenou hodnotu; pokud na list druhého typu, vracíme hodnotu vzniklou dosazením onoho čísla z $< 0, 1 >$ do "funkce distribuce".

Při výběru s odebráním vybraného prvku se situace komplikuje o to, že musíme vhodným způsobem přepočítat některé dělící hodnoty. Tyto hodnoty jsou však jen na cestě k danému prvku a tak je časová složitost stále O(log n).

Nutno poznamenat, že struktura je navržena hlavně pro práci s listy prvního typu a tak některé operace, jako *výběr s odebráním* nebo *nalezení prvku s maximální pravděpodobností*, se chovají pro distribuce obsahující listy druhého typu jinak, než by bylo formálně korektní. Konkrétně: Odebíráme vždy celý list stromu; a při nalezení prvku s maximální pravděpodobností hledáme ve skutečnosti list s maximální pravděpodobností, přičemž v případě listu typu dva vracíme reprezentanta pro 0.5.

(Činíme tak z toho důvodu, že zde mícháme dohromady diskrétní a spojité rozdělení, přičemž nám jde ale spíše o ty diskrétní vlastnosti a ty spojité se nám hodí jen specifickým způsobem.)

Nyní se podívejme na centrální část GP-algoritmu; na šlechtění.



Vezmeme-li hodnoty FF pro jednotlivé jedince počáteční populace jako pravděpodobnosti, můžeme jednoduše zkonstruovat pravděpodobnostní rozložení jedinců populace. Od teď se na populaci budeme spíše než jako na seznam jedinců koukat jako na distribuci jedinců.

Z tohoto rozložení pak vybíráme dvojice jedinců (rodičů) které křížíme. (Dále bychom mohli provádět mutaci, tu však pro jednoduchost zatím nemáme, i když mutace je částečně použita v křížení, viz níže.) Jejich děti pak jdou do další generace. Jedno místo je také rezervováno pro nejlepšího jedince z generace, aby jsme si v každé generaci drželi nejlepší dosavadní řešení.

Nejzajímavější na šlechtící fázi je určitě křížení. Oproti jednoduchému křížení v klasickém GP zde musíme hlídat dvě věci: Typy a proměnné.

Mějme dva termy, které chceme zkřížit. Křížení probíhá tak, že najdeme v každém z obou termů nějaký podterm a tyto prohodíme. Podrobněji: Odtržením podtermu od termu dostaneme *"zbytek"* termu; křížení odpovídá výběru podtermů v obou termech a následnému prohození jejich zbytků. Aby term zůstal dobře otypován, musí být vybrány podtermy stejného typu. To realizujeme tak, že ze všech dvojic typově kompatibilních podtermů jednu náhodně vybereme.

Druhou věcí, kterou musíme pří křížení ošetřit, jsou volné proměnné v prohazovaných podtermech. Každý jedinec je kombinátor a my tuto vlastnost nechceme křížením pokazit, proto budeme nově vzniklým volným proměnným hledat svázání v jeho nově nabytém *"zbytku"* - tzn. projdeme tento nový zbytek a budeme v něm hledat lambda abstrakce se stejným typem proměnné, jaký má naše inkriminovaná volná proměnná. Pokud jich najdeme víc, vybereme jednu náhodně. Následně v tomto podtermu nahradíme všechny výskyty této proměnné touto zvolenou proměnnou. Pokud žádná taková proměnná není, pak se musíme této volné proměnné zbavit. V současném prototypu to řeším tak, že vygenerujeme nový kombinátor stejného typu jako má ona nepohodlná volná proměnná a tímto termem nahradíme všechny výskyty nepohodlné proměnné. Tento nový kombinátor vygenerujeme stejnou metodou, kterou generujeme počáteční populaci. Vlastně se jedná o formu *mutace*. Nakonec ještě u nově vzniklých jedinců zunikátníme jména všech proměnných (tzn. přejmenujeme proměnné tak, aby každé jméno proměnné bylo vázáno právě jednou lambda abstrakcí).

Nyní se vrátíme k *enviromentu* (neboli sadě stavebních bloků) a zamyslíme se nad tím, zda by nebylo vhodné rozšířit ho o nějaké další dodatečné informace. Jednou z mých počátečních motivací pro chuť začlenit typy do GP byla představa univerzální sady stavebních bloků společná pro všechny problémy - o takovéto univerzální sadě pak už můžeme přemýšlet jako o *jazyku*. Čím však enviroment obsahuje víc stavebních bloků, tím je prostor prohledávaných programů větší a stavební bloky nesouvisející s řešením podstatně zhoršují kvalitu populace. Na druhou stranu otypovanost stavebních bloků mnoho slepých uliček utne. Stavební bloky sebou nesou dodatečnou informaci v podobě typů, která umožňuje tvořit smysluplnější programy (než kdybychom například všechna data kódovali přirozenými čísly a pak uvažovali jen stavební bloky operující nad přirozenými čísly). Otázka, kterou si kladu ve svém dalším průzkumu, zní: Jaké další informace (a jakým způsobem) by šlo k enviromentu přidat, aby to umožnilo tvorbu smysluplnějších/relevantnějších jedinců?

- Enviroment je seznam stavebních bloků. První možnost přidání informace je vzít místo seznamu *distribuci*, čili chápat enviroment jako pravděpodobnostní rozložení stavebních bloků.

- Program se skládá z *stavebních bloků* a *konstruktů jazyka*. Konstruktům jazyka v našem případě odpovídají jednotlivá "redukční pravidla nad rozdělanými termy" (po jednom pro *[Axiom]*, $[E^{\rightarrow}]$ a $[I^{\rightarrow}]$). Není toto dělení nějakým způsobem umělé? Vždyť *[Axiom]* vlastně realizuje samotné dosazování stavebních bloků. Dalším přidáním informace do enviromentu je přesunutí konstruktů jazyka z "vnitřku" dokazovací jednotky do enviromentu. Čímž se zároveň podporuje představa enviromentu jakožto *jazyka*. Čili mezi *stavební bloky*, připočítáme i konstrukty jazyka.

- Další otázka zní: Jak šikovně *zadávat* rozložení stavebních bloků? Prvoplánové řešení

je vzít jednoduše seznam typu [(*Stavební_blok,Pravděpodobnost*)]. Myslím, že o něco šikovnější je do věci zavést hierarchii: Seznam nahradíme stromem, kde listy představují *Stavební_bloky* a hrany z nelistových vrcholu jsou ohodnoceny nezáporným reálným číslem. Každý nelistový vrchol stromu představuje "bod rozhodnutí" a jednotlivé hrany představují "možnosti" jak se v tomto vrcholu rozhodnout, ohodnocení hrany pak představuje (nenormalizovanou) pravděpodobnost výběru daného rozhodnutí.

- Další krok spočívá v pokusu o zobecnění předchozí stromové struktury, tak aby nám mimo jiné umožnila elegantně vkládat do programů náhodné číselné hodnoty (v dosavadním uchopení jsme s čísli zacházeli přes kostrbaté konstanty). K tomu jsem navrhnul strukturu *DecTree* inspirovanou *Rozhodovacím stromem*. Popis provedu značně neformálně: Na rozdíl od klasického rozhodovacího stromu, kde je výsledkem jediná "odpověď", u DecTree je výsledkem "distribuce odpovědí". Listy stromu odpovídají "odpovědím". Nelistové vrcholy "bodům rozhodnutí". Hrany jsou ohodnocené (typicky číslem).
  Nacházíme se v situaci, kdy stromu pokládáme "otázku". Touto otázkou je v našem případě "rozdělaný term" a odpovědí na tuto otázku nám bude distribuce "možných vzniklých rozdělaných termů použitím přepisovacích pravidel odpovídajících stavebním blokům/konstruktům jazyka". Princip spočívá v tom, že do kořene pošleme číslo 1. V každém nelistovém vrcholu dlí rozhodovací mechanizmus (v různých vrcholech mohou být různé mechanizmy), který na základě "otázky" a "ohodnocení hran", které z něj vedou, rozhodne jak rozdělit to co do něj přišlo mezi jednotlivé hrany. Z těchto hran to přiteče do dceřiných vrcholů, kde se celá situace opakuje. Dokud číslo nedoteče do listu, kde představuje (nenormalizovanou) pravděpodobnost této odpovědi.

Čili abych to trochu shrnul: Mluvili jsme o tom, jak rozšířit enviroment o další informaci. Heslovitě:

- spíš než seznam distribuci
- neoddělovat konstrukty jazyka od sady stavebních bloků
- distribuci zadávat hierarchicky
- předchozí tři body realizovat pomocí "variace na rozhodovací stromy"

V tuto chvíli už možná vyplouvá na povrch další skrytá touha, totiž zkusit tyto "variace na rozhodovací stromy" šlechtit pomocí GP algoritmu. Na to si však ještě nějakou dobu nechám zajít chuť - dokud kód a sada testovacích příkladů nebude mnohem vyzrálejší.

Nyní se dostáváme k druhému způsobu generování počáteční populace, skrze nově uchopený enviroment. Nyní je enviroment zadán pomocí DecTree, který nám pro rozdělaný term dá distribuci rozdělaných termů. Z této distribuce máme možnost vybírat buď klasicky nebo s odebráním prvku. Ty prvky, které vybereme odpovídají směrům, kterými se vydáme v prohledávání prostoru A* algoritmem. Položme si následující otázku: Jak pomocí nového enviromentu simulovat chování dokazovače se starým enviromentem? Pokud z *Dist* budeme postupně vybírat prvky s odebráním, do té doby dokud z ní nevybereme všechno - pak nepřeskočíme žádnou možnost a dokazovač se bude chovat stejně jako ten starý. Nové chování tedy spočívá v tom, že nebudeme prozkoumávat úplně všechny možnosti systematicky.

Novému dokazovači přibývá nový parametr, dvojice z množiny <0,1>×<0,1>. První číslo odpovídá tomu, kolik "následníků" současného rozdělaného termu dál navštívíme (počet je

dán relativně v zhledem k max. počtu následníků). Druhé číslo je pravděpodobnost toho, zda výběr z distribuce bude s odebráním nebo bez. Čili pro (1,1) se chová jako starý dokazovač (až na drobné rozdíly, jako je lehce přeházené pořadí v kterém budou jednotlivé programy generovány). Pro (0.75,0.5) máme jen 75% celkového počtu následníků, přičemž se navíc někteří mohou i opakovat víckrát, protože je 50% šance že se bude vybírat bez odebrání.

V současném prototypu je implementován GP-systém, který si bere jako vstupní parametr "dokazovač". Bohužel je v implementaci někde chyba a GP-systém funguje jen pro starší typ dokazovače (novější funguje obstojně sám, ale někde je patrně schovaná muška v interakci GP-algoritmu a dokazovače).

V rychlosti popišme z jakých modulů se první prototyp v tuto chvíli skládá:

- *Base* - Účelem modulu Base je definice typu TTerm a s ním spolupracujících funkcí. TTerm neboli "oTypovaný TERM" můžeme nazvat "základním typem celého programu", protože reprezentuje termy/programy, které "šlechtíme". Používáme ho jak při generování termů, tak při evaluaci (alespoň zatím).
  Přesnější charakterizací než "otypovaný term" je "rozpracovaný otypovaný term", v tom smyslu, jako je pro lidského programátora rozpracovaný zdrojový kód nehotového programu. Ve zkratce se jedná o lambda term rozšířený o "položku typ" a o "možnost být rozdělaný".

- *(Util* - Util obsahuje obecné funkce nad standardními typy.)

- *Dist* - Dist představuje implementaci distribuce (pravděpodobnostního rozložení). Popsáno dost podorbně výše.

- *Evaluation* - Modul starající se o vyhodnocování termů.
  Jakým způsobem se nyní TTermy evaluují a jakým způsobem to chci změnit v budoucnu: Oproti klasickému lambda kalkulu rozlišujeme mezi "proměnnou" a "hodnotou". Hodnoty odpovídají vestavěným kombinátorům - tyto kombinátory jsou (když to trochu zjednodušíme) Hasskellovské hodnoty "zabalené" pomocí knihovny Data.Dynamic umožňující obejít typový systém Haskellu (o zjednodušení zde mluvíme protože existují i jiné typy hodnot/kombinátorů, které ale pro nás teď nemá smysl uvažovat). TTermy vyhodnocujeme líně. Kýžené líné vyhodnocování je však podstatně komplikováno právě zmíněnými vestavěnými kombinátory: do Haskellovské funkce musíme dosadit haskellovskou hodnotu, čili ne TTerm. Tento problém řešíme tak, že slevíme z úplné lenosti a tím vzniká ne úplně pěkný hybrid. U tohoto hybridu jsem ale prozatím zůstal, protože jsem považoval za lepší nejdříve udělat "jakž-takž" fungující prototyp celého systému a až po této fázi začít novou iteraci vývoje a celou věc přepsat "bytelně". (To má za následek nepěkné vedlejší efekty, jako např. neohrabané podchycení kombinátoru "if" atd.)
  Čím tedy nahradit toho současného hybrida? V zásadě mám dva kandidáty přičemž první z nich je můj jasný oblíbenec, ale je pracnější:

  - (A) Opustit snahu "exploitovat" haskellovské hodnoty a místo toho sáhnout k napsaní si pořádného interpretu podle [Jones, S. P. (1991). Implementing Functional Languages]
  - (B) Použít knihovnu Hint (což je Runtime Haskell interpreter (GHC API wrapper)).

Pro favorizaci možnosti (A) mám několik důvodů: Při evaluaci považuji za velice užitečné mít k dispozici možnost omezit maximální počet beta redukcí případně jiné podobné možnosti. To by mělo jít (snad) dobře, díky tomu že zmíněná kniha je podrobný tutoriál, čili budu mít v ruce kód, kterému budu rozumět a tím pádem ho budu moci hezky rozšiřovat. Další důvod je chuť proniknout do tajů funkcionálních překladačů.

Na obhajobu (B) lze říct, že by kýžené funkcionality šlo dosáhnout nějakými časovými omezeními na maximální dobu běhu. Ale přijde mi, že z toho jde takový ten pocit "na komára s kanónem". Nejobjektivnějším rozhřešením by nejspíš bylo otestovat, co je rychlejší (což neumím odhadnout).

Ještě se nabízí možnost (C) slevit z lenosti a pak by to možná šlo udělat pomocí současné metody dostatečně konzistentní, tuhle metodu ale zatím moc nezvažuji jako finální řešení, spíš jako mezifázi.

- *Parser* - OSTUDA !! :) TODO : předělat do Parsecu
- *DecTree* - Variace na rozhodovací strom, použitá k uchopení nového enviromentu - rozšířeného o další informace krom typu.
- *Enviroments* - Obsahuje definice enviromentů, v nich obsažených programů a jejich typů.
- *Prover* - Zajišťuje generování počátečních populací.
- *GP* - Jádro genetického algoritmu. Výpočet probíhá ve *State* monádě.
- *Ant* - Jediný netriviální testCase, problém převzatý z Kozy: šlechtíme program jednoduchého mravence tak, aby sežral co nejvíc jídla. Pohybuje se po čtvercové síti a pozná, když je přímo před ním jídlo, umí se otáčet doleva, doprava a pohnout dopředu. Dále pak je k dispozici příkaz pro zřetězení víc příkazů. Zajímavé je to tím, že v zájmu zachování problému přesně tak jak byl originálně bylo nutné simulovat stavový výpočet.
- *Main* - Obsahuje testCases.

Co budu dělat dál?

- Rád bych si "přistřihnul křidýlka" a pečlivě implementovat Kozův klasický GP-algoritmus (společně se zásobou jeho testCasů), ale v řeči těch mých konceptů - tzn. snažit se udělat klasický přístup jako speciální případ toho mého. (Abych dokázal, že ty moje koncepty nejsou samoúčelné a abych měl s čím porovnávat).
- Udělat pořádnou evaluaci termů.
- Zrobustnit kód.
- Přečíst si články vyšlé o genetickém programování v kombinací s typovánými funkcionálními jazyky.