Typed Genetic Programming in Lambda Calculus

T. Křen

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. TODO

Introduction

Genetic programming (GP) is an AI technique, which falls into broader category of evolutionary algorithms — mataheuristic search algorithms inspired by biological evolution by natural selection. GP was developed by John Koza in 1992 Koza [1992]. I dare say many people perceive its beauty in the fact that GP is a computer program constructing other computer programs with desired properties by breeding them. It is also pretty successful technique in a number of areas [Koza, 2003]. And perhaps that is why GP has become very popular.

Early attempts to enhance the GP approach with the concept of types include the seminal work [Montana, 1995] where the ideas from Ada programming language were used to define a so-called strongly typed GP. Usage of types naturally opens door to enriching S-expressions, the traditional GP representation of individuals, with concepts from lambda calculus, which is a simple yet powerful functional, mathematical and programming language extensively used in type theory [Yu, 2001].

One of the motivations for using lambda calculus is that it is backed by a considerable theoretical background. From this background we can utilize several useful concepts such as reductions, normalization or abstraction elimination, and use them to enhance the standard GP algorithm.

(- ještě něco k tomu že LC je cool pro TGP (taky z článků) ?)

- že důležitý je navrhnout generování a gene ops. že gener de přímoč do mutace.

The rest of the paper is organized as follows: TODO

Preliminaries

In this section, several notions necessary to build a typed GP based on lambda calculus are introduced.

Genetic programming

A problem to be solved is given to GP in a form of *fitness function*. Fitness function is a function which takes computer program as its input and returns numerical value called *fitness* as output. The bigger fitness of a computer program is, the better solution of a problem. GP maintains a collection of computer programs called *population*. A member of population is called *individual*. By running GP algorithm evolution of those individuals is performed.

Individuals are computer program *expressions* kept as *syntactic trees*. Basically those trees are rooted trees with a function symbol in each internal node and with constant symbol or variable symbol in each leaf node. Number of child nodes for each internal node corresponds to the number of arguments of a function whose symbol is in that node.

Another crucial input besides fitness function is a collection of *building symbols*. It is a collection of symbols (accompanied with an information about number of arguments). Those symbols are used to construct trees representing individuals.

Let us describe GP algorithm briefly. At the beginning, initial population is generated from building blocks randomly. A step of GP algorithm is stochastic transformation of the current population into the next population.

This step consists of two sub steps:

- Selection of *parents* for individuals of the next population based on the fitness. The bigger fitness of an individual of the current population is, the better chance of success being selected as parent it has.
- Application of genetic operators (such as *crossover*,*reproduction* and *mutation*) on parent individuals producing new individuals of the next population.

This transformation is repeatedly applied for a predefined number of steps (which is called number of *generations*) or until some predefined criterion is met.

Lambda calculus

Let us describe a programming language, in which the GP algorithm generates individual programs — the so called λ-terms.

Definition 1 *Let V be infinite countable set of variable names. Let C be set of constant names, $V \cap C = \emptyset$. Then Λ is set of λ-terms defined inductively as follows.*

$$\begin{aligned} x \in V \cup C &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (M\ N) \in \Lambda \qquad \textit{(Function application)} \\ x \in V, M \in \Lambda &\Rightarrow (\lambda\, x\, .\, M) \in \Lambda \qquad \textit{(λ-abstraction)} \end{aligned}$$

Function application and λ-*abstraction* are concepts well known from common programming languages. For example, in JavaScript, $(M\ N)$ translates to expression M`(`N`)` and $(\lambda\, x\, .\, M)$ translates to expression `function(`x`){return` M`;}`. In other words, the function application corresponds to the act of supplying a function with an argument, and the λ-abstraction is equivalent to *anonymous function*[1]. To ensure better readability, $M_1\ M_2\ M_3\ \ldots\ M_n$ will be an abbreviation for $(\ldots((M_1\ M_2)\ M_3)\ \ldots\ M_n)$. And $\lambda\, x_1 \ldots x_n\, .\, M$ for $(\lambda\, x_1\, .\, \ldots (\lambda\, x_n\, .\, M)\ldots)$.

Reductions

In order to perform computation, there must be some mechanism for term evaluation. In λ-calculus there is a β-reduction procedure for this reason.

A term of a form $(\lambda\, x\, .\, M)N$ is called β-*redex*. A β-redex can be β-reduced to term $M[x := N]$. This fact is written as *relation* $\rightarrow_\beta$ of those two terms:

$$(\lambda\, x\, .\, M)N \rightarrow_\beta M[x := N] \tag{1}$$

It is also possible to reduce *subterm* β-*redexes* which can be formally stated as:

$$\begin{aligned} P \rightarrow_\beta Q &\Rightarrow (R\ P) \rightarrow_\beta (R\ Q) \\ P \rightarrow_\beta Q &\Rightarrow (P\ R) \rightarrow_\beta (Q\ R) \\ P \rightarrow_\beta Q &\Rightarrow \lambda\, x\, .\, P \rightarrow_\beta \lambda\, x\, .\, Q \end{aligned}$$

In other words, β-reduction is the process of insertion of arguments supplied to a function into its body.

Another useful relations are $\twoheadrightarrow_\beta$ and $=_\beta$ defined as follows.

1. (a) $M \twoheadrightarrow_\beta M$
   (b) $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$
   (c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$

[1]Apart from JavaScript, anonymous functions are common e.g. in Python and Ruby, they were recently introduced to C++, and they are expected to be supported in Java 8.

:

2. (a) $M \twoheadrightarrow_\beta N \Rightarrow M =_\beta N$
   (b) $M =_\beta N \Rightarrow N =_\beta M$
   (c) $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$

We read those relations as follows.

1. $M \twoheadrightarrow_\beta N$ — "M β-reduces to N."
2. $M \rightarrow_\beta N$ — "M β-reduces to N in one step."
3. $M =_\beta N$ — "M is β-convertible to N."

Similarly as for β-reduction we can define η-reduction except that now it is defined as follows.

$$(\lambda\, x\,.\,(M\ x)) \rightarrow_\eta M \qquad \textbf{if } x \notin FV(M)$$

Analogically, a term of a form $(\lambda\, x\,.\,(M\ x))$ is called *η-redex*.

Relation[2] $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$. Similarly as for $\twoheadrightarrow_\beta$ and $=_\beta$ we can define relations $\twoheadrightarrow_\eta$, $=_\eta$,$\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$.

The η^{-1}-reduction (also called η-expansion) is the reduction converse to η-reduction. It is defined as follows.

$$M \rightarrow_{\eta^{-1}} (\lambda\, x\,.\,(M\ x)) \qquad \textbf{if } x \notin FV(M)$$

Normal forms

Definition 2

1. *A λ-term is a β-normal form (β-nf) if it does not have a β-redex as subterm.*
2. *A λ-term M has a β-nf if $M =_\beta N$ and N is a β-nf.*

A normal form may be thought of as a result of a term evaluation. Normalization of the term M is the process of finding normal form of M. Similarly we can define *η-nf* and *$\beta\eta$-nf*.

Types

A λ-term as described above corresponds to a program expression with no type information included. Now we will describe *types* (or *type terms*).

Definition 3 *Let A be set of atomic type names. Then $\mathbb{T}$ is set of types inductively defined as follows.*

$$\alpha \in A \Rightarrow \alpha \in \mathbb{T}$$
$$\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \rightarrow \tau) \in \mathbb{T}$$

Type $\sigma \rightarrow \tau$ is type for functions taking as input something of a type σ and returning as output something of a type τ. $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$ is an abbreviation for $\tau_1 \rightarrow (\tau_2 \rightarrow (\ldots \rightarrow (\tau_{n-1} \rightarrow \tau_n)\ldots))$. The system called *simply typed λ-calculus* is now easily obtained by combining the previously defined *λ-terms* and *types* together.

Definition 4

1. *Let Λ be set of λ-terms. Let $\mathbb{T}$ be set of types. A statement $M : \sigma$ is a pair $(M, \sigma) \in \Lambda \times \mathbb{T}$. Statement $M : \sigma$ is vocalized as "M has type σ". The term M is called the subject of the statement $M : \sigma$.*

[2]Relation $R = \{\ (a, b) \mid a\ R\ b\ \}$.

2. *A declaration is a statement* $x : \sigma$ *where* $x \in V \cup C$.

3. *A context is set of declarations with distinct variables as subjects.*

Context is a basic type theoretic concept suitable as a typed alternative for terminal, and function set in standard GP. Notation $\Gamma, x : \sigma$ denotes $\Gamma \cup \{(x : \sigma)\}$ such that Γ does not contain any declaration with x as subject. We also write $x : \sigma \in \Gamma$ instead of $(x, \sigma) \in \Gamma$.

Definition 5 *A statement* $M : \sigma$ *is derivable from context* Γ *(notation* $\Gamma \vdash M : \sigma$*) if it can be produced by the following rules.*

$$\begin{aligned} x : \sigma \in \Gamma \;&\Rightarrow\; \Gamma \vdash x : \sigma \\ \Gamma \vdash M : \sigma \to \tau \;,\; \Gamma \vdash N : \sigma \;&\Rightarrow\; \Gamma \vdash (M\ N) : \tau \\ \Gamma, x : \sigma \vdash M : \tau \;&\Rightarrow\; \Gamma \vdash (\lambda\, x\,.\, M) : \sigma \to \tau \end{aligned}$$

Our goal in term generation is to produce terms M for a given pair $\langle \tau; \Gamma \rangle$ such that for each M is $\Gamma \vdash M : \tau$.

Long normal form

Definition 6 *Let* $\Gamma \vdash M : \sigma$ *where* $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha, n \geq 0$.

1. *Then* M *is in long normal form (lnf) if following conditions are satisfied.*

   (a) M *is term of the form* $\lambda\, x_1 \ldots x_n\,.\, f\ M_1\ \ldots\ M_m$
   (specially for $n = 0$, M *is term of the form* f*).*

   (b) *Each* M_i *is in lnf.*

2. M *has a lnf if* $M =_{\beta\eta} N$ *and* N *is in lnf.*

As is shown in Barendregt et al. [2010], *lnf* has following nice properties.

Proposition 1 *If* M *has a* β*-nf, then it also has an unique lnf, which is also its unique* $\beta\eta^{-1}$*-nf.*

Proposition 2 *Every* B *in* β*-nf has a lnf* L *such that* $L \twoheadrightarrow_\eta B$.

Abstraction elimination

Abstraction elimination is a process of transforming an arbitrary λ-term into λ-term that contains no lambda abstractions and no bound variables. The newly produced λ-term may contain function applications, free symbols from former λ-term and some new symbols standing for combinators **S**, **K** and **I**.

Those combinators are defined as:

$$\begin{aligned} \mathbf{S} &= \lambda\, f\, g\, x\,.\, f\, x\, (g\, x) \\ \mathbf{K} &= \lambda\, x\, y\,.\, x \\ \mathbf{I} &= \lambda\, x\,.\, x \end{aligned}$$

:

Let us describe transformation AE performing this process.

$$
\begin{aligned}
\mathrm{AE}[x] &= x \\
\mathrm{AE}[\,(M\,N)\,] &= (\mathrm{AE}[M]\ \ \mathrm{AE}[N]) \\
\mathrm{AE}[\lambda\,x\,.\,x] &= \mathbf{I} \\
\mathrm{AE}[\lambda\,x\,.\,M] &= (\mathbf{K}\ \ \mathrm{AE}[M])\ \textbf{if}\ x \notin \mathrm{FV}(M) \\
\mathrm{AE}[\lambda\,x\,.\,(\lambda\,y\,.\,M)] &= \mathrm{AE}[\lambda\,x\,.\ \mathrm{AE}[\lambda\,y\,.\,M]]\ \textbf{if}\ x \in \mathrm{FV}(M) \\
\mathrm{AE}[\lambda\,x\,.\,(M\,N)] &= (\mathbf{S}\ \ \mathrm{AE}[\lambda\,x\,.\,M]\ \ \mathrm{AE}[\lambda\,x\,.\,N]) \\
&\quad \textbf{if}\ x \in \mathrm{FV}(M) \vee\ \ x \in \mathrm{FV}(N)
\end{aligned}
$$

As is stated in [Peyton Jones, 1987], the biggest disadvantage of this technique is that the translated term is often much larger than in its lambda form — the size of the translated term can be proportional to the square of the size of the original term. But the advantage is also tempting — no need to deal with variables and lambda abstractions.

The algorithm presented here is a simple version of this process. More optimized version (used in our *hybrid* crossover described below), by means of the size of resulting term and its time performance is presented in [Peyton Jones, 1987].

- HM - z papírů, wiki
- Typová unifikace
- Type classes - z papírů (asi ne tak formální)
- naznačit celou hierarchii, řict že na vršku je jakoby DTT.

Related work

Yu [2001] presents a GP system utilizing polymorphic higher-order functions[3] and lambda abstractions. Important point of interest in this work is use of `foldr`[4] function as a tool for *implicit recursion*, i.e. recursion without explicit recursive calls. The terminal set for constructing lambda abstraction subtrees is limited to use only constants and variables of that particular lambda abstraction, i.e., outer variables are not allowed to be used as terminals in this work. This is significant difference from our approach since we permit all well-typed normalized λ-terms. From this difference also comes different crossover operation. We focus more on term generating process; their term generation is performed in a similar way as the standard one, whereas our term generation also tries to utilize techniques of systematic enumeration.

Briggs and O'Neill [2008] present technique utilizing typed GP with combinators. The difference between approach presented in this work and our approach is that in this work terms are generated straight from *library* of combinators and no lambda abstractions are used. They are using more general polymorphic type system than us – the Hindley–Milner type system. They also discuss the properties of exhaustive enumeration of terms and compare it with GP search. They also present interesting concept of *Generalized genetic operator* based on term generation.

Binard and Felty [2008] use even stronger type system (*System F*). But with increasing power of the type system comes increasing difficulty of term generation. For this reason evolution in this work takes interesting and nonstandard shape (fitness is associated with *genes* which are evolved together with *species* which together participate in creation of individuals). This differs from our approach, which tries to be generalization of the standard GPKoza [1992].

---

[3] Higher-order function takes another function as an input parameter.

[4] In the functional programming language Haskell `foldr` can be defined as:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

In contrast with above mentioned works our approach uses very simple type system (simply typed lambda calculus) and concentrates on process of generation able to generate all possible well-typed normalized lambda terms. In order to do so we use technique based on *inhabitation machines* described by Barendregt Barendregt et al. [2010].

Todo, ještě neco?

Our current approach

Generating

Our approach to λ-term generating is based on technique briefly described in [Barendregt et al., 2010], which generates well-typed λ-terms in their long normal form. We use this technique to perform a systematic exhaustive enumeration of λ-terms in their long normal form in order from the smallest to the largest. We use well known *A* algorithm* [Russell and Norvig, 2010] for this task. A* is used to search in a given state space for a goal state. It finds the optimal solution (in our case the smallest term) and uses "advising" heuristic function. It maintains a priority queue to organize states yet to be explored. Initially this queue contains only the initial state.

Our state space to search in is the space of unfinished λ-terms. The initial state is the unfinished term $\langle \tau; \Gamma \rangle$, where τ is the desired type of terms to be generated and Γ is the context representing the set of building symbols to be used in construction of terms (it corresponds to the set $T \cup F$ in standard GP enriched with types). The process of determining successors of a state described below is designed so it constructs well-typed λ-terms and omits no λ-term in its long normal form. A state is considered a goal state if it contains no unfinished leaf, i.e., it is a finished λ-term.

Our generating method is based on simple modification of the standard A*, which we call *forgetful A**. This modification consist in additional parameter for the A* algorithm – the *search strategy*. It is a simple filtration function (along with initialization procedure) that is given the set of all successors of the state that is being examined and returns a subset of this input. This subset is added to the priority queue to be further explored. In this way the search space may be reduced as the filtration function may *forget* some successors. If the queue becomes empty before the desired number of λ-terms is generated, then the initial state is inserted to the queue and the process continues. For the standard A* this would be meaningless, but since our A* is forgetful this kind of restart makes sense.

A* keeps a priority queue of states during the generation process, on the other hand the *ramped half-and-half method*, the standard GP algorithm for generating individuals, keeps only one individual which is gradually constructed. This behavior is easily achieved by use of suitable search strategy that returns subset consisting of only one successor. The systematic search is obtained by search strategy that returns whole input set. Our novel *geometric strategy* can be understood as point somewhere between those two extremes.

A remarkable benefit of parameterizing the generating method by a search strategy taking the form of a simple filtration function is that such a function can be expressed by functional language in a very economical way. Take for instance a search strategy that behaves same as the *geometric* except that when it comes to situation where all the elements are filtered out it acts as the *ramped half-and-half*. Such *strategy composition* is easily describable as a higher order function. Thus comes to mind the possibility of using our system to come up with new search strategies on its own.

Crossover

Our system utilizes generalization of the standard tree-swapping crossover operator. Two main concerns with swapping typed subtrees are types and free variables. Well-typed offspring is obtained by swapping only subtrees of the same type. Only subtrees with corresponding

counterpart in the second parent are randomly chosen from. More interesting problem lies in free variables, which may cause trouble if swapped somewhere where it is suddenly not bounded. In order to circumvent this difficulty we utilize technique called *abstraction elimination*[Peyton Jones, 1987] that transforms an arbitrary λ-term into λ-term that contains no lambda abstractions and no bound variables. After the initial population is generated, it is transformed by abstraction elimination. Another possible transformation taking place after initialization is η-normalization shortening rather long long normal form into shorter $\beta\eta$-normal form. Another performance enhancing transformation is option of using "applicative" tree representation (coming directly from inductive definition of λ-terms) instead of more traditional S-expression representation. Favorable properties of applicative tree representation are also reported in [Yu and Clack, 1998].

Future work

Jak pojmout podstatu článku?

Řekl bych, že vtip je v použití typovejch tříd na hendlování typovýho systému TGP.

Tahle pasáž je imho klíčová a tak bych jí napsal jako první, podle noušnů v ní obsaženejch bych pak upravil preliminaries a akordingly to zmínil v related work na konci (něco jako, na rozdíl od kanaďanů my se chceme skusit vydat na cestu k zobecnění HM tak, že využijem typeclasses : výhody videíme v tom, že se prakticky využívaj haskellu a jejich použití výrazně neztěžuje inferenční proces, naopak do něj přirozeně pasuje) nekde dyštak zmínit že to jsou fikaný javovský interfacy

Zásadním rozhodnutím při budování systému pro typované GP je volba underlying typového sytému. Současné přístupy k typovanému GP nad lambda calculem nejčastěji používají hidley-milnerův typový system citovat yu, kombinatory nebo experimentují s Systemem F, kterýžto je zobecněním HM. Alternativní přístup k obohacení HM would be enriching the HM type system with the type classes.

A type class is a type system construct that supports ad-hoc polymorphism in a mathematically elegant way. The concept of the type class first appeared in the Haskell programming language [Morris, 2013] and was originally designed as a way of implementing overloaded arithmetic and equality operators in a systematic fashion [Wadler and Blott, 1989].

Basically a type class is a predicate over types. Let us demonstrate this notion on the simple example of a type class, the `Eq` type class for handling the equality operator.

Funkci `intMember` zjištující zda číslo x je v seznamu čísel xs můžeme zapsat jako:

```
intMember ::  Int -> [Int] -> Bool
intMember y [] = False
intMember y (x:xs) = (x == y) || intMember y xs
```

Abychom nemusel mít funkci pro každý typ zvlášť, hodilo by se mít zobecněnou funkci `member ::  a -> [a] -> Bool`, navíc bychom chtěli abychom její definici mohli převzít z intMember. Tomu brání pouze použití equality operatoru (==). Abychom tuto definici mohli použít, potřebujeme aby nad typem a byla definována rovnost. Tento fakt můžeme vyjádřit jako predikát `Eq a`. V typu funkce member se tato dodatečná podmínka projeví jako `member ::  (Eq a) => a -> [a] -> Bool`. Zbívá však dořešit jakým způsobem se typová třída Eq definuje. Nejjednoduším způsobem definice by bylo(footnote doopravdy je složitější ale to je jedno):

```
class Eq a where
(==) ::  a -> a -> Bool
```

Ještě zbývá poslední věc a to jak říct systému, že nějaký typ splňuje predikát Eq. Řekněme,

že to chceme udělat pro typ Int a že máme k dispozici funkci eqInt :: Int -¿ Int -¿ Bool která implemetuje operator rovnosti nad cisly. Pak stačí deklarovat následující:

```
instance Eq Int where
(==) = eqInt
```

Co kdybychom ale chtěli definovat Eq nad parametrickým typem, řekněme nad stromem. Jak se vyhneme tomu abychom nemuseli deklarovat instanci pro každý typ seznamu zvlášť? K tomu nám poslouží následující notace:

```
instance (Eq a) => Eq (Tree a) where
Leaf a == Leaf b = a == b
(Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
_ == _ = False
```

v rychlosti okomentovat a zmínit že obecně těch předpokladů tam může bejt víc a jakou to má notaci. řict že pro nás je z typového hlediska podstatná ta deklarace a to co následuje po where už je otázka implementace (možná to dohrotit tak, že tu implementaci ...-ovat.)

Jak to převést do GP? Konkrétněji jak generovat programy v HM rozšířeném o TC? Pro generování bez TC je potřeba zadat požadovaný typ a mn. stavebních symbolu (s informací o jejich typech) tj. context. Pro HM s TC musíme navíc zahrnout deklarace tříd a instancí.

Deklarace tříd udává jednak podmínky které musí splnit deklarace instancí. Navíc pokud používáme TC Eq, je patrně nasnadě aby součástí contextu byly funkce, které daná třída definuje. E.g., pro smysluplné použití třídy Eq je potřeba aby součástí vstupniho contextu bylo `(==) :: (Eq a) => a -> a -> Bool`.

Hlavní novinku do vstupu generujícího algoritmu zajišťují deklarace instancí. Množina instancí nám v záseadě přidává informaci o struktuře typů, nad kterými operují funkce z kontextu.

Pro účely generování nás zajíma jen "info z prvního řádku", v zasadě máme dva typy instancí jednoduchou a s předpoklady. Zásadním faktem je, že tyto informace o instancích můžeme přímočaře vyjádřit jako logické programy citovat něco kde se to dělá.

`instance` $P\ T$ `where ...` odpovídá faktu $P(T)$.

`instance (`$P_1\ T_1$`,...,`$P_n\ T_n$`) => `$P\ T$ `where ...` odpovídá hornovské klauzuli $P(T) \leftarrow P_1(T_1), \ldots, P_n(T_n)$.

Pro přehlednost jsme uvedli predikátové symboli jen s jedním argumentem, přímočaře to však můžeme zobecnit pro predikáty libovolné arity.

Odpověď na otázku zda daný typ je instancí nějaké TC zodpovíme odpovídajícím dotazem pro logický program odpovídající naší množině instancí.

- dodat tuto LP složku je na programátarovy, GP se pak stará o generování. TC jsou ale obecné konstrukty které jde jednoduše používat napříč problémy (užití Systému F naopak nechává vše na GP, minimum na programátorovi)

- neboť LP tur completní tak v zasadě můžeme naprogramovat libovolnej predikát

- blackboxing - můžeme se vysrat na to pracně definovat predikáty v LP, stačí když se budou zvenku chovat tak jako by byli a vevnitř mohou bejt napsaný třeba v javě

- praktický příklad takovýho blackboxingu - např plánování. ukazuje, že tyto dvě problem solving approaches are uzce connected - využití např že se muže GP inspirovat technikama z Planování, naopak zas že type theory může dát plánování expresivní jazyk / novej pohled na pro škatulkizaci různejch pojmů co odpovídaj nějakejm věcem co v GP jsou přirozeně (fce víc argumentů..., lamb abstr...)

- nakonec (/na začatek) zmínit že máme i další future worky ale že vybíráme tento konkrétní

:

Starší poznámky ...

- HM .. místo = dá unifikaci, a bez letu.
(možná tam dát dk, spiš ne)
- uvod k hm+tc 1-3 věty.

hinley milner + type classes

příklady
(možná i ten s planovanim, dle času a místa)
(metaevoluce)

Problémy

- sada benchmarku - citovat ten šit co sem našel
- trhy

. - vyhody: propojenej uzel uloh (slechteni manazera, modelu, konstruktera stroju,), je to vypocetni model, na trh muzeme koukat jako na . - meta model - výpočetní model co má to co optimalizuje zakodovany jako komoditu to co . - navazuje na spolupráci mezi plánováním a

Conclusion

Todo.
v referencích změnit toho barendrechta na toho z roku 2013

Acknowledgments. ???

References

Barendregt, H., Dekkers, W., and Statman, R., *Lambda Calculus With Types*, Cambridge University Press, 2010.

Binard, F. and Felty, A., Genetic programming with polymorphic types and higher-order functions, in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pp. 1187–1194, ACM, 2008.

Briggs, F. and O'Neill, M., Functional genetic programming and exhaustive program search with combinator expressions, *International Journal of Knowledge-Based and Intelligent Engineering Systems*, *12*, 47–68, URL `http://iospress.metapress.com/content/u6l4j13p67w66370/`, 2008.

Koza, J., *Genetic Programming IV*, Genetic Programming IV: Routine Human-competitive Machine Intelligence, Kluwer Academic Publishers, URL `http://books.google.cz/books?id=vMaVhoI-hVUC`, 2003.

Koza, J. R., *Genetic programming: on the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, USA, 1992.

Montana, D., Strongly typed genetic programming, *Evolutionary computation*, *3*, 199–230, 1995.

Morris, J. G., *Type Classes and Instance Chains: A Relational Approach*, Ph.D. thesis, Portland State University, 2013.

Peyton Jones, S. L., *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

Russell, S. J. and Norvig, P., *Artificial Intelligence - A Modern Approach (3rd Ed.)*, Prentice Hall, 2010.

Wadler, P. and Blott, S., How to make ad-hoc polymorphism less ad hoc, in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 60–76, ACM, 1989.

:

Yu, T., Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction, *Genetic Programming and Evolvable Machines*, *2*, 345–380, 2001.

Yu, T. and Clack, C., PolyGP: A polymorphic genetic programming system in haskell, *Genetic Programming*, *98*, 1998.