Typed Genetic Programming in Lambda Calculus

T. Křen

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

Abstract. TODO
[Článek zatím obsahuje i české poznámky, které jsou hodně hala bala a neformální. A celkově je to ještě dost neučesané. Dost jsem zápasil s nedostatkem místem a hodně toho dal pryč, protože jsem dal radši přednost podrobnějšímu popisu klíčových pojmů..]

Introduction

Genetic programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [Koza, 1992, 2003]. 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. The use of types naturally leads to enriching S-expressions, the traditional GP representation of individuals, with concepts from *lambda calculus*, which is simple yet powerful functional, mathematical, and programming language extensively used in type theory. Such attempts has shown to be successful 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 typed GP (TGP).

In this paper we compare typed lambda calculus approaches to GP known to us with approach we are suggesting as possible alternative. The crucial decision behind the design of a TGP system is the choice of the underlying *type system*. The majority of such TGP systems uses *Hindley–Milner type system (HM)*. There are also experiments with more powerful type systems such as System F. An alternative approach we proposse is to use *HM* enriching with the *type classes*.

The rest of the paper is organized as follows: TODO

Preliminaries

In this section several useful notions are described.

Genetic programming

A problem to be solved is given to GP in the form of a *fitness function*. A fitness function is a function which takes a computer program as its input and returns a numerical value called *fitness* as an 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 a population is called *individual*. By running GP algorithm evolution of those individuals is performed. Individuals are computer program *expressions* kept as *syntactic trees*. 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; or in the case of the typed GP, this information is the type of the symbol). Those symbols are used to construct trees representing individuals.

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 $(\lambda\, x_1\,.\,\ldots(\lambda\, x_n\,.\, M)\ldots)$.

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 $M[x := N]$. It is also possible to reduce *subterm β-redexes*. In other words, β-reduction is the process of insertion of arguments supplied to a function into its body.

Simply typed lambda calculus

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

Definition 2 *Let A be set of atomic type names. Then $\mathbb{T}$ is set of types inductively defined by the following two rules: (1) $\alpha \in A \Rightarrow \alpha \in \mathbb{T}$, and (2) $\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \to \tau) \in \mathbb{T}$.*

Type $\sigma \to \tau$ is type for functions taking as input something of a type σ and returning as output something of a type τ. $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ is an abbreviation for $\tau_1 \to (\tau_2 \to (\ldots \to (\tau_{n-1} \to \tau_n)\ldots))$. Such "chain of arrows" types simulate types of functions with multiple (here $n-1$) inputs. This technique is called *currying*. One can grasp the trick by considering $(f\ M_1\ \ldots\ M_n)$ as shorthand for $f(M_1, \ldots, M_n)$ and by observing that functions with such types have terms of the form $\lambda x_1 x_2 \ldots x_n\,.\, M$. The *type system* called *simply typed λ-calculus (STLC)* is now easily obtained by combining the previously defined *λ-terms* and *types* together.

Definition 3 *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 $M : \sigma$. A declaration is a statement $x : \sigma$ where $x \in V \cup C$. 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 4 *A statement $M\colon \sigma$ is derivable from a context Γ (notation $\Gamma \vdash M\colon \sigma$) if it can be produced by the following rules.*

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

[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.

Those rules are called *inference rules* (see the similarity with the *definition 1*). A type system is defined by a collection of inference rules. When generating terms (e.g., during population initialization) our goal is to produce terms M for a given pair $\langle \tau; \Gamma \rangle$ such that for each M is $\Gamma \vdash M : \tau$ (where τ is the desired type of generated individuals and Γ is the set of *building blocks*). More complex type systems (e.g. Hindley–Milner or System F) can be defined similarly by specifying their set of inference rules, but (for the sake of brevity) we prefer to continue the overview in a less formal fashion.

Hindley–Milner type system

Hindley–Milner type system [Hindley, 1969] is a generalization of the simply typed lambda calculus extended by two concepts: parametric types and type variables (and with the `let` expression, but we omit its discussion since it is less important for the type related matter).

Parametric types are types that take other type(s) as parameter. E.g., a list (or a tree) of values of the same type. A type variable is a type that may stand for an arbitrary type. Those two concepts play nicely together, we can see this by observing following simple example. The function `head` which returns the first element of a list. This function has type `a -> List a`, where `a` is a type variable and `List` is a parametric type (taking another type as parameter, in this case it is a type variable `a`). In STLC we must define the head function for every concrete instance of the List, whereas in the Hindley–Milner we can define them all at once.

System F

System F [?] can be seen as a further generalization of both STLC and Hindley–Milner. The concept of type variables brings the question of variable quantification. Hindley–Milner uses implicit general quantification. System F is about making things explicit. This holds both for placing variable quantification symbols ($\forall$) explicitly inside the type terms, and for need to explicitly place the type term into the polymorphic function in order to cast it to the specific instance with no type variables. This for example enables possibility of generating "anonymous" type definitions[2]. This rise in power brings the disadvantage of the substantially more computationally complex term generating procedure.

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:

Let `isMemberInt : Int -> [Int] -> Bool` be a function performing check whether an integer (the first argument) is a meber of a list of integers (the second argument). And let us assume that this function in its definition uses the equality operator:

```
isMemberInt y [] = False
isMemberInt y (x:xs) = (x == y) || isMemberInt y xs
```

In order to overcame the need to have for each type (such as `Int`) a specific instance of the function it would be nice to have a way to specific a function such as `member : a -> [a] -> Bool`, but there is a trouble with the equality operator used in the definition. The `==` is defined only on some types. The fact that an equality operator is defined for type `T` is denoted by use of the *predicate* `Eq` by writing `(Eq T)`. We can specify this additional requirement in the type as `member : (Eq a) => a -> [a] -> Bool`.

---

[2] E.g., the standard `List a = Cons a (List a) | Nil` can be expressed "anonymously" as $\forall\alpha\forall\beta : (\alpha \to \beta \to \beta) \to \beta \to \beta$.

Such a predicate is called *type class* and is defined in the following way:

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

In order to state the fact that a certain concrete type (e.g. `Int`) is a type for which the predicate holds we write the following declaration (and suppose that `eqInt :: Int -> Int -> Bool` is our implementation of integer equality):

```
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 (where we omit the implementation part since we are interested in the signature):

```
instance (Eq a) => Eq (Tree a) where ...
```

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

Related work

Todo, přepsat aby to pasovalo na porovnání s type classama; připadně to porovnání udělat celý až v následující sekci o our approach

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 GP[Koza, 1992].

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. [2013].

Our current approach

---

[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 our previous work jsme se rozhodli pro přístup: nejdřív vzít nejjednoduší typovej systém a v něm navrhnout generování a gen operace tak aby jednak byly zobecnění std GP, a jednak aby byly schopný generovat libovolnej term (modulo NF), ne jen nějakou jednodušší podmnožinu.

Pokud se to tam vejde tak zmínit, že HM lze z STLC získat tak že (zjednodušeně řečeno) na vhodných mistech změníme porovnání rovnosti typových termů za unifikaci typových termů. (V tý nasledující sekci pak k tomu navíc přidat, že se k unifikaci přidává běh logickýho programu, který je ale nutno pouštět po krokách tak, aby byly na stejný urovni kroky generujícího algoritmu a kroky běhu toho LP (tzn že rozpracovaný term muže být několik kroků A* v tom samém termu ale s jiným stádiem dopočítanosti logického programu pracujícího na doplnění dalšího symbolu).)

Konkretní popis mojí generující (a křížící) metody zatim zakomentován protože to zabírá moc místa, nejdřív dopsat zbytek a pak vzít co se vejde, pokud vůbec něco..

Future work

Jak TC 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 zásadě 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.

- říct že sou 2 použití - jednak v původnim smyslu pro ad-hoc polymorphism - ale taky exploitačnim způsobem kde nám vůbec nejde o implementační složku (která může být prázdná), ale jde nám jen o LP věci. Názornym příkladem takovýho použití je zavedení čísel (pozor neplest s termovými čísly) do typovýho systému.

- taková čísla se nám mohou hodit např při práci se seznamy pevné délky.

- 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 - nemusíme se trápit s tím pracně definovat predikáty v LP, stačí když se budou zvenku chovat tak jako by byli a vevnitř mohou bejt napsaný třeba v javě

- když zbude místo (což se asi nestane) tak tmínit příklad blackboxingu - plánování vyjádřený vjako specialní případ typovanýho GP. Spíš asi než takle komplikovanej příklad dát ale na víc míst par drobnejch příkladů využití

- určitě shrnout výhody HM+TC approache...

Conclusion

Todo nebo spojit s future work do jedný subsekce.
v referencích změnit toho barendrechta na toho z roku 2013

References

Barendregt, H., Dekkers, W., and Statman, R., *Lambda calculus with types*, Cambridge University Press, 2013.

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, 2008.

Hindley, R., The principal type-scheme of an object in combinatory logic, *Transactions of the american mathematical society*, pp. 29–60, 1969.

Koza, J., *Genetic Programming IV*, Genetic Programming IV: Routine Human-competitive Machine Intelligence, Kluwer Academic Publishers, 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.

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.