

# **Utilization of Reductions and Abstraction Elimination in Typed Genetic Programming**

Tomáš Křen, Roman Neruda

Charles University, Faculty of Mathematics and Physics &  
Institute of Computer Science, Academy of Sciences of the Czech Republic

# Types in GP

- Types help us overcome the *closure requirement*.
  - No longer need for “everything fitting into everything”.
- But they also establish new requirements
  - e.g. function arguments must obey type requirements...
  - These requirements also makes the programs more reasonable,
  - and reduce the search space.

# Lambda calculus

- Simple yet powerful (mathematical) functional programming language
- It uses anonymous functions very often.
- Roughly speaking:

s-expressions + anonymous functions = lambda calculus



*x may occur in this subtree...*

$\lambda$  *<var-name>* . *<body-expr>*

aka

**function(<var-name>){ return <body-expr>; }**

# Benefits of using Functional programming for GP

- Complex and/or general programming constructs can be described as higher-order functions
  - Example 1: **foldr** function for implicit recursion
    - Another view: generalized for cycle
    - Body of for cycle corresponds to anonymous function given to foldr as argument.
  - Example 2: Explicit application for ADF-like behavior
    - `function(f,x){ return f(x) }`
- Types provide rigorous way to talk about (sub)programs and to enforce constraints.

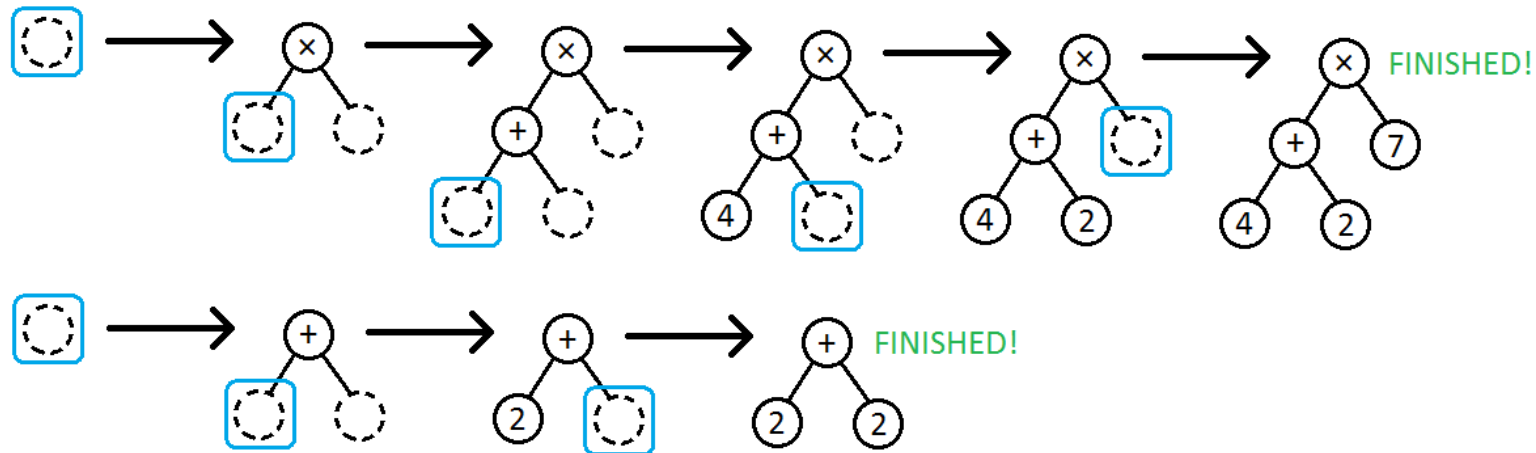
# Related work

- Yu : PolyGP
  - We use more general term representation which is not restricting the use of outer local variables in the body of an anonymous function.
  - Hindley-Milner type system
- Briggs and O'Neill : Purely combinator approach
  - purely combinator approach (no anonymous functions, no local variables)
  - We have the advantage of reducing the size of the search space in the term generation phase.
  - Hindley-Milner type system
- Binard and Felty
  - Even stronger type system (System F)  $\rightarrow$  non-standard algorithm
  - Genes (i.e. subtrees) have fitness

# Our approach to typed GP over lambda calculus

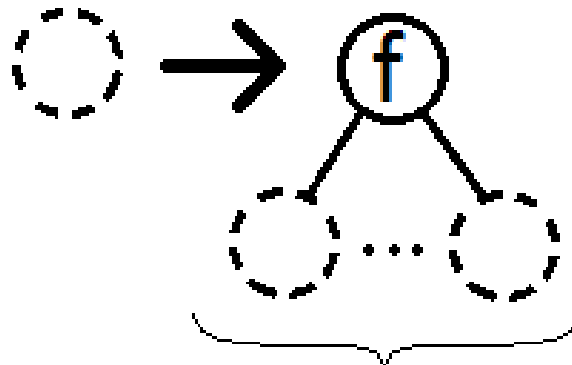
- Simple type system (Simply typed lambda calculus)
  - with future generalizations in mind (H-M, H-M with type classes)
- ..but complete generating
- Anonymous functions
- Proper generalization of the standard GP

# Tree generating terminology



- **Expansion**

- In each step an **unfinished node** is replaced by a more specific tree:



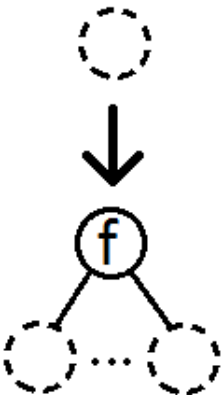
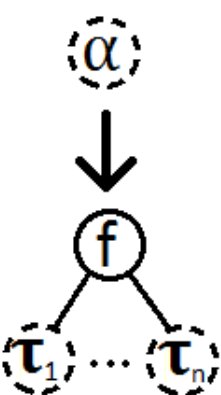
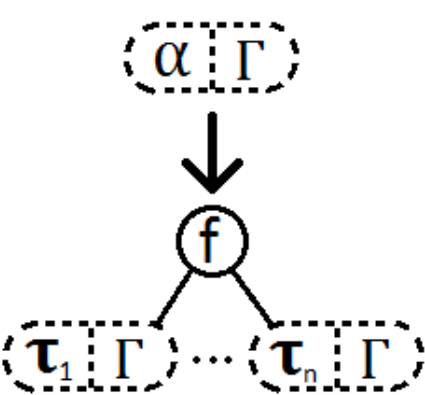
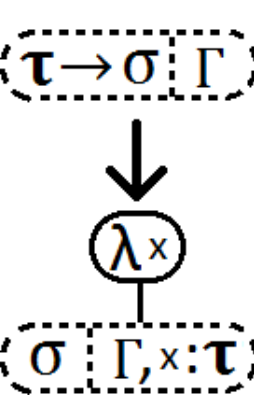
*0 or more unfinished nodes for subtrees*

# Local context (aka $\Gamma$ )

- Set of symbol names accompanied with types
- “ $\mathbf{T} \cup \mathbf{F}$  is an initial/global context”
- We can add local variables to a context



# Expansions for Simply typed lambda calculus

	No types	Types, no contexts	Simply typed lambda calculus	
Unfinished node	$\circ$	$\alpha$	$\tau \mid \Gamma$	
Expansion(s)		 $f : \underbrace{\tau_1 \rightarrow \dots \rightarrow \tau_n}_{\text{inputs types}} \rightarrow \underbrace{\alpha}_{\text{ouput type}}$	<i>atomic types:</i>  $(f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \alpha) \in \Gamma$	<i>function types:</i> 

# Generating and reductions

- Reduction: transformation of one term into another

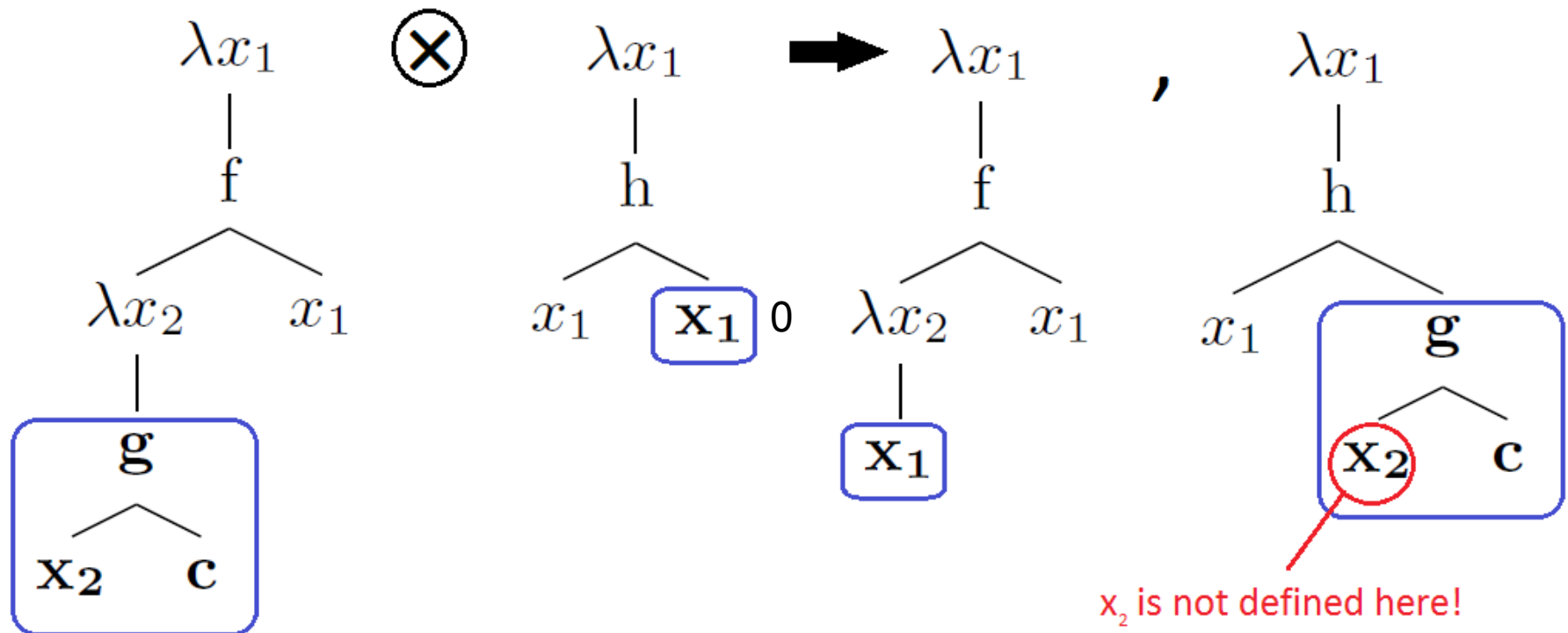
*beta reduction:*  $(\lambda x . M)N \rightarrow_{\beta} M[x := N]$

*eta reduction:*  $(\lambda x . (M \ x)) \rightarrow_{\eta} M \quad \text{if } x \notin FV(M)$

- From reduction arises notion of term equivalence (relative to that reduction)
- Normal form: term that can't be reduced
- Described method generates *long normal form (Inf)*
  - Each *Inf* term represent whole *beta-eta-equivalence class*
  - **This reduces the search space**

# Crossover operator for lambda terms

- Generalization of simple tree swapping crossover
- We need to swap subtrees with a same type
  - ..but that is simple
- Local variables cause the trouble!



# Abstraction elimination

- An algorithm for getting rid of local variables and anonymous function
  - **Input:** an arbitrary lambda term
  - **Output:** equivalent S-expression that may contain additional new function nodes **S**, **K** and **I** which are defined as:

$$S = \lambda f g x . f x (g x)$$

**"function(f,g,x){ return f(x, g(x)) }"**

$$K = \lambda x y . x$$

**i.e. "function(x,y){ return x }"**

$$I = \lambda x . x$$

**"function(x){ return x }"**

# Hybrid crossover

- Each generated term is transformed by abstraction elimination
- All terms are typed S-expressions
- So now we only need to swap subtrees with the same type
- *Hybrid* because
  - lambda term representation during generation phase
    - Advantage: reduced search space during generation phase
  - Pure combinator representation during the rest
- Possible disadvantage: up to quadratic increase in term size

# Unpacking crossover

- All terms are kept in small  $\beta\eta$ -normal form
- ...and transformed right before crossover
- After the tree swapping both children are again normalized
- So the quadratic increase is only temporary

# Experiment

- We compare performance of *hybrid* and *unpacking* crossover.
- Even parity problem
  - Similar function and terminal set as in *PolyGP* by T. Yu

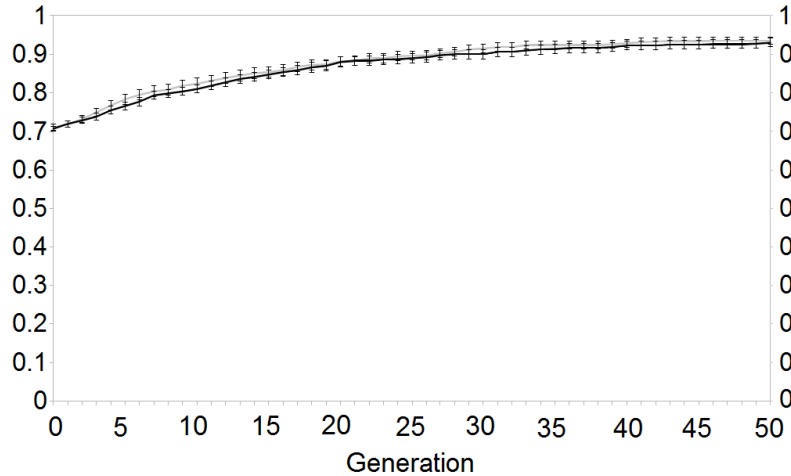
$$\boxed{\tau_0 \mid \Gamma_0}$$

$$\tau_0 = [Bool] \rightarrow Bool$$

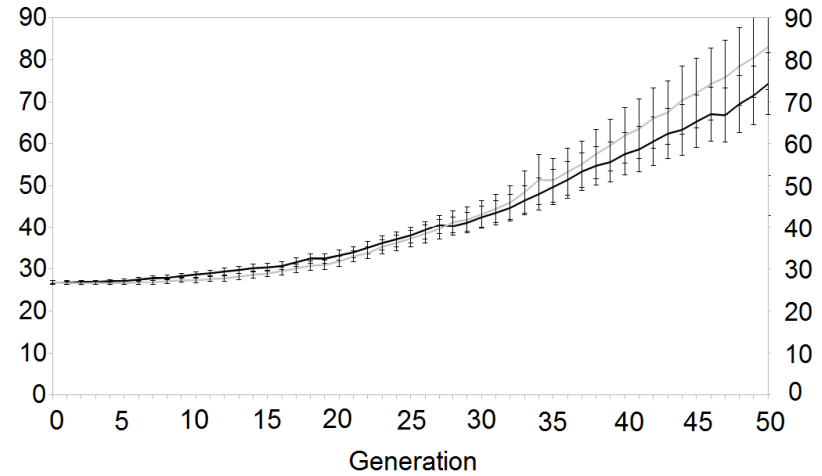
$$\Gamma_0 = \{and : Bool \rightarrow Bool \rightarrow Bool, \\ or : Bool \rightarrow Bool \rightarrow Bool, \\ nand : Bool \rightarrow Bool \rightarrow Bool, \\ nor : Bool \rightarrow Bool \rightarrow Bool, \\ foldr : (Bool \rightarrow Bool \rightarrow Bool) \\ \rightarrow Bool \rightarrow [Bool] \rightarrow Bool, \\ head' : [Bool] \rightarrow Bool, \\ tail' : [Bool] \rightarrow [Bool]\}$$

# Results

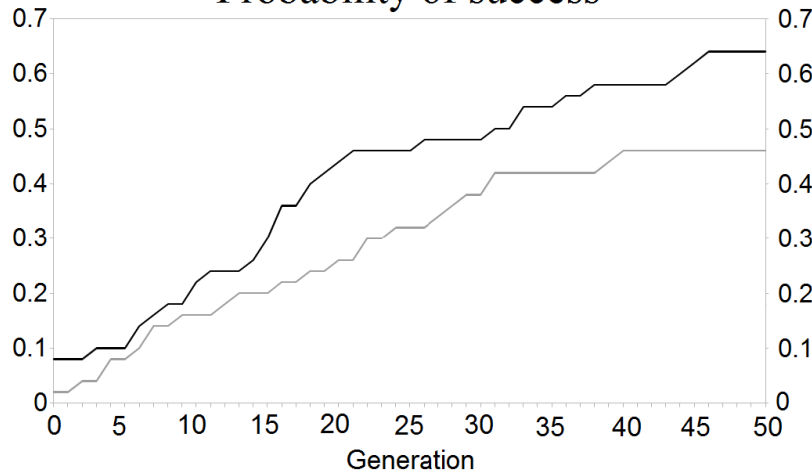
Fitness of the best individual



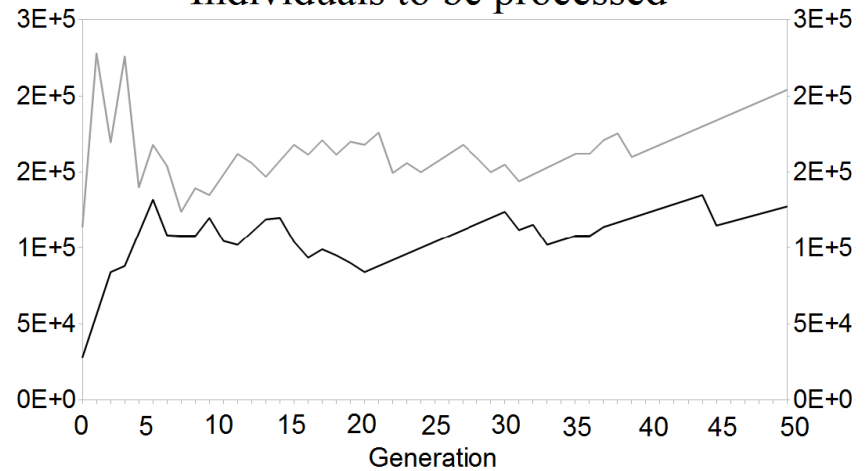
Average term size



Probability of success



Individuals to be processed

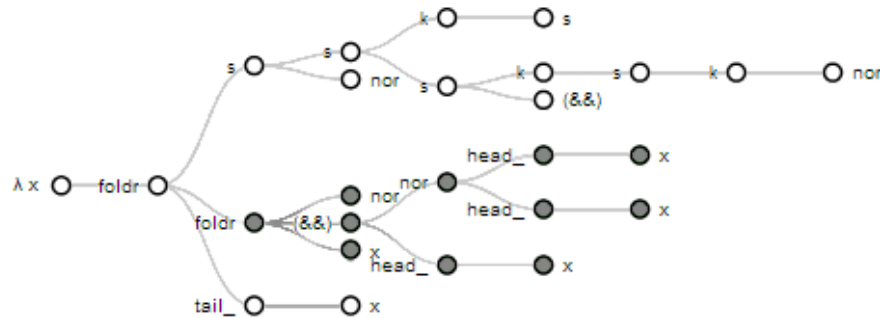


Unpacking crossover  
Hybrid crossover

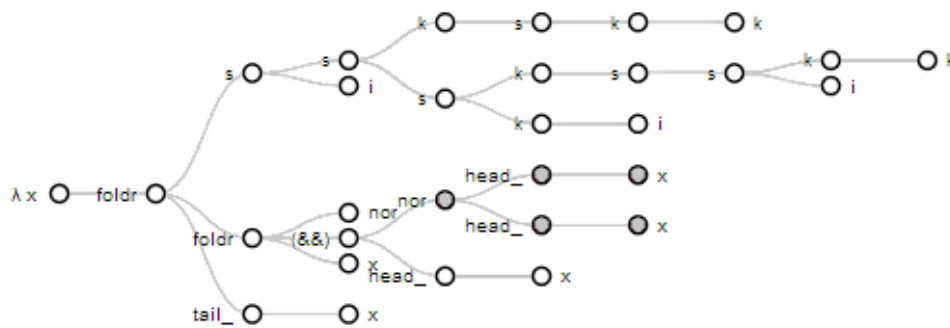
Times: **388** minutes  
**33** minutes



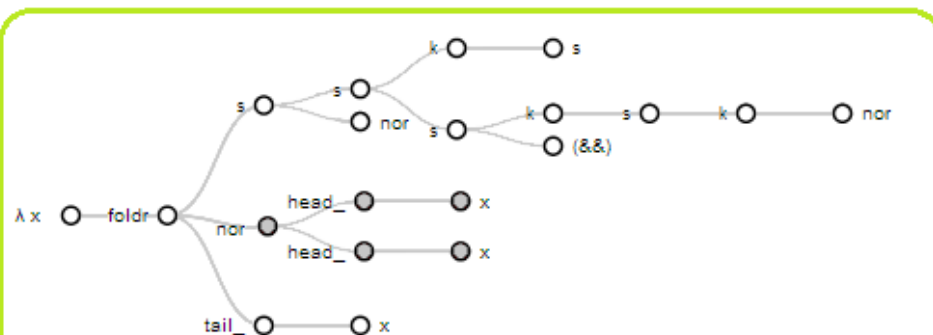
# Results



Parent #1, fitness: 0.8125



Parent #2, fitness: 0.5



Child #1, fitness: 1

$$\lambda x . foldr (\mathbf{S}(\mathbf{S}(\mathbf{K} \mathbf{S})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K} \text{ nor}))))\text{and}))\text{nor}) \\ (\text{nor} (\text{head}' x) (\text{head}' x)) (\text{tail}' x)$$

$=_{\beta\eta}$

$$\lambda x . foldr (\lambda y z . \text{nor} (\text{and } y z) (\text{nor } y z)) \\ (\text{nor} (\text{head}' x) (\text{head}' x)) (\text{tail}' x)$$

Which is equivalent to:

$$\lambda x . foldr \text{ xor } (\text{not} (\text{head}' x)) (\text{tail}' x)$$

GP approach	$I(M, i, z)$
PolyGP	14,000
<b>Our approach (hybrid)</b>	<b>28,000</b>
GP with Combinators	58,616
GP with Iteration	60,000
<b>Our approach (unpacking)</b>	<b>114,000</b>
Generic GP	220,000
OOGP	680,000
GP with ADFs	1,440,000

Results comparison

# Conclusions

- How to generate lambda terms in *Inf*
  - Why? It reduces search space
- How to crossover terms containing local variables using abstraction elimination technique
  - Hybrid vs. Unpacking : Hybrid wins
- Future work
  - More experiments
  - More elaborate analysis (Price's covariance for crossover,...)
  - Stronger type systems (H-M, H-M with type classes)

# Thank you for your attention!

Any questions?

[tomkren@gmail.com](mailto:tomkren@gmail.com)  
[roman@cs.cas.cz](mailto:roman@cs.cas.cz)