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

Presentation outline

- Types & lambda calculus : Why to use them in GP?
- Related work
- Generating and reductions
- Crossover operators for lambda terms : problems with local variables
- Experiment

Types in GP

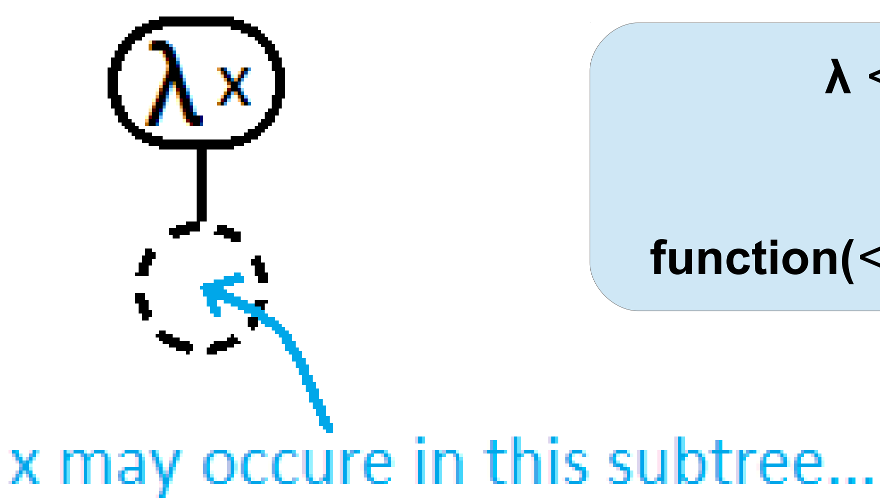
- Types help us overcome the *closure requirement*.
  - No longer need for “everything fitting into everything”.

- But they also establish new requirements
  - e.g. function arguments must obey type requirements...
  - These requirements make 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



λ *<var-name>* **.** *<body-expr>*

aka

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

Benefits of using Functional programing 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.
- Types provide rigorous way to talk about (sub)programs and to enforce constraints.

Related work

- Yu : PolyGP
  - uses lambda term representation, but restricts 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
  - Hindley-Milner type system
- Binard and Felty
  - Even stronger type system (System F) → non-standard algorithm
  - Genes (i.e. subtrees) have fitness

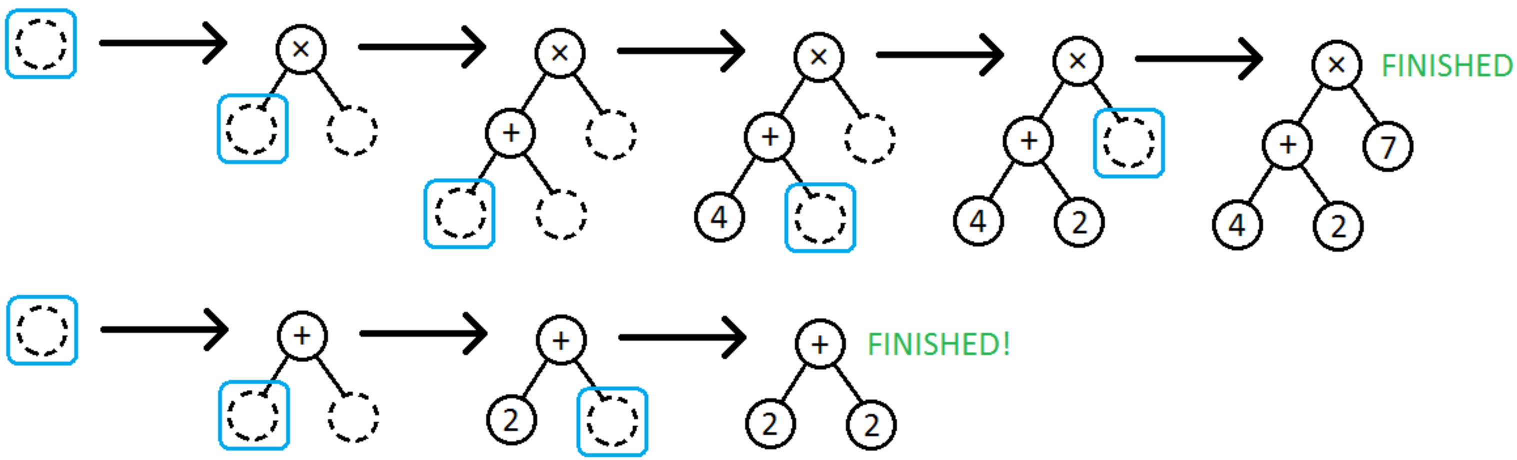
Our approach to typed GP over lambda calculus

- Simple type system (Simply typed lambda calculus)
  - with future generalizations in mind (H-M, H-M with type classes)
- ..but complete generating

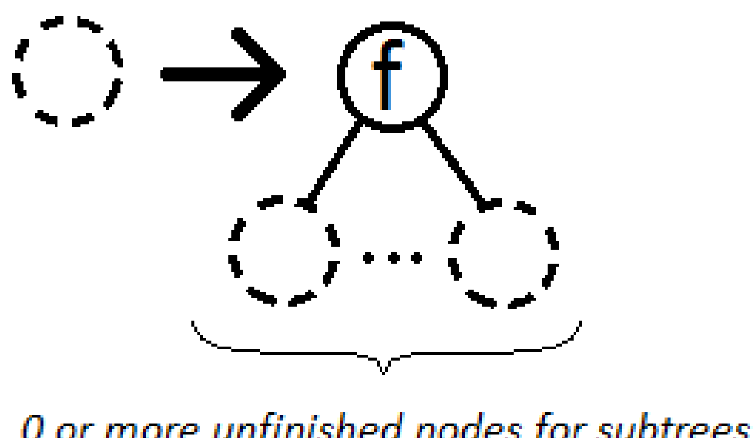
- Anonymous functions

- Generalization of the standard GP

Tree generating terminology



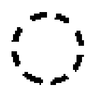
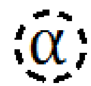
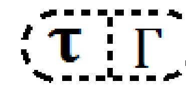
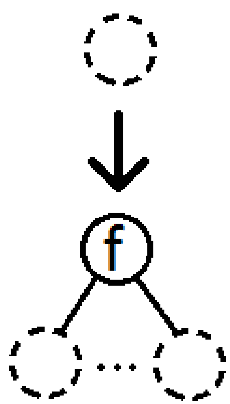
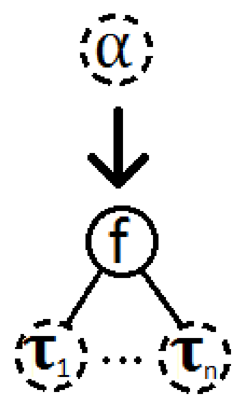
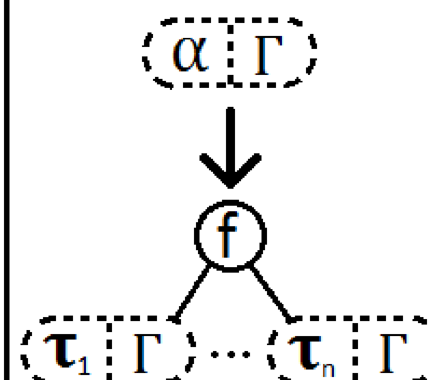
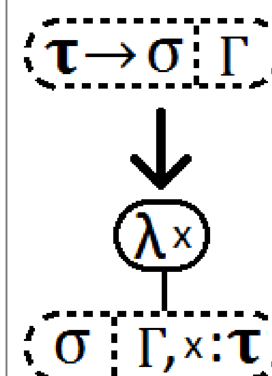
- ***Expansion***
  - In each step an ***unfinished node*** is replaced by a more specific tree:



Local context (aka Γ)

- 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				
Expansion(s)		 $f : \underbrace{\tau_1 \to \cdots \to \tau_n}_{\text{inputs types}} \to \underbrace{\alpha}_{\text{ouput type}}$	*atomic types:*  $(f : \tau_1 \to \cdots \to \tau_n \to \alpha) \in \Gamma$	*function types:* 

Generating and reductions

- Reduction: transformation of one term into another

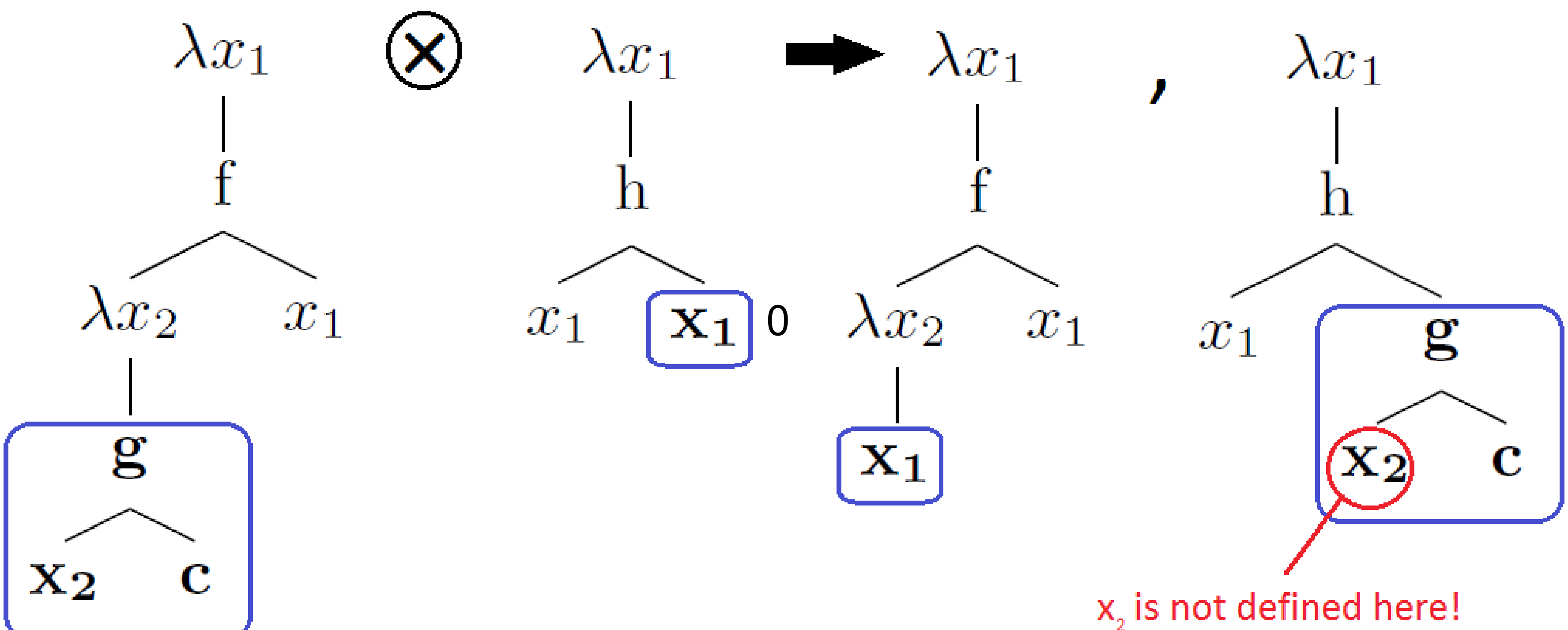
beta reduction: $(\lambda x\,.\,M)N \rightarrow_\beta M[x := N]$

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

- 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 (lnf)*
  - Each *lnf* 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 (*with no local variables and no anonymous functions)* that may contain additional new function nodes **S**,**K** and **I** which are defined as:

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

i.e.

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

"**function**(x,y){ **return** 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 βη-normal form
- ...and transformed right before crossover
- After the tree swapping both children are again normalized

- So the quadratic increase is only temporary

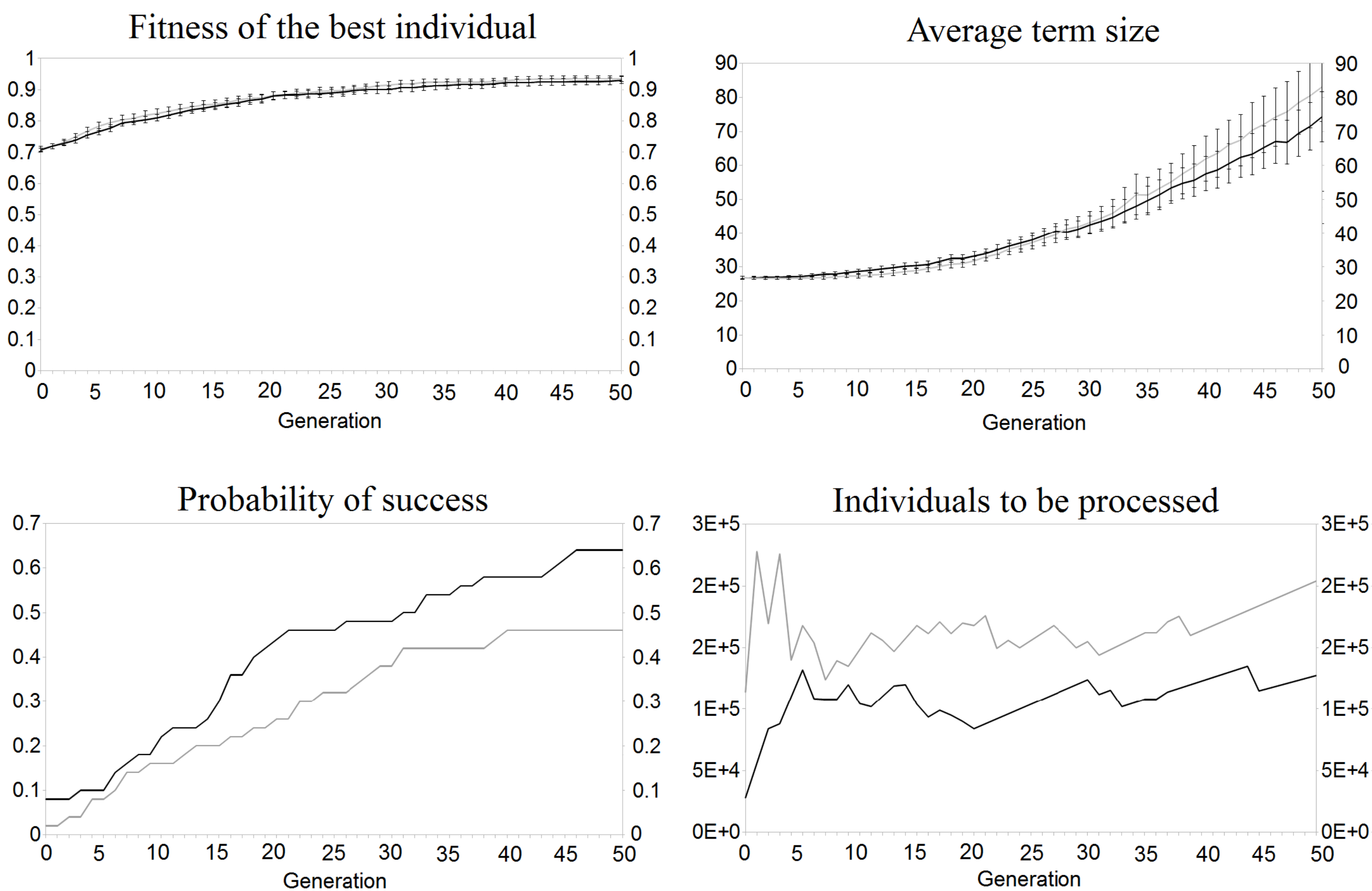
Experiment

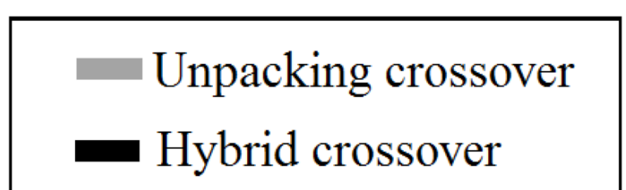
- We compare performance of *hybrid* and *unpacking* crossover.
- Even parity problem
  - Similar function and terminal set as in *PolyGP* by *T. Yu*

$\tau_0 \mid \Gamma_0$

$$
\begin{aligned}
\tau_0 &= [Bool] \to Bool \\
\Gamma_0 &= \{and : Bool \to Bool \to Bool, \\
& or : Bool \to Bool \to Bool, \\
& nand : Bool \to Bool \to Bool, \\
& nor : Bool \to Bool \to Bool, \\
& foldr : (Bool \to Bool \to Bool) \\
& \quad \to Bool \to [Bool] \to Bool, \\
& head' : [Bool] \to Bool, \\
& tail' : [Bool] \to [Bool]\}
\end{aligned}
$$

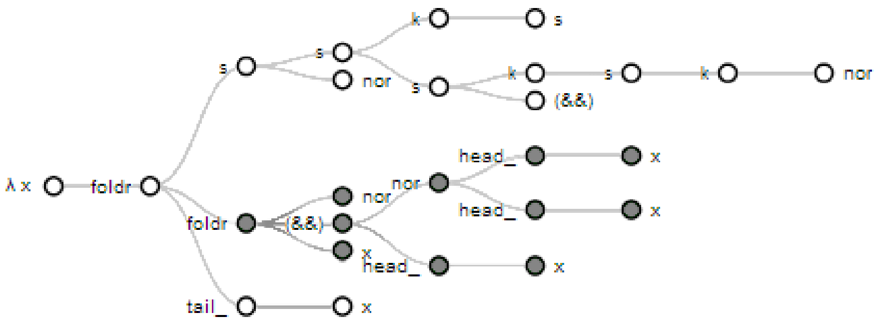
Results



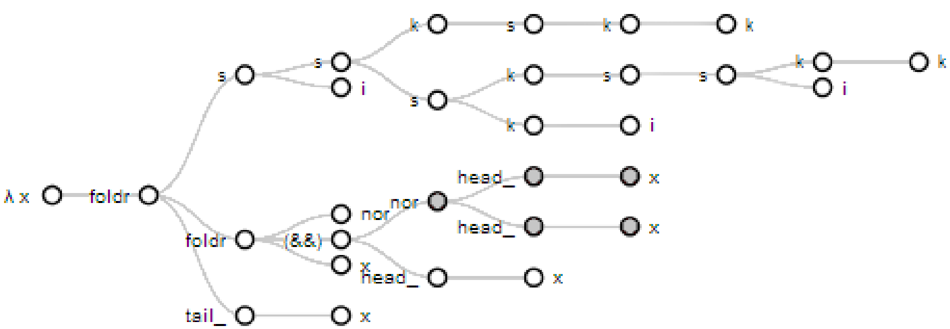


Times: **388** minutes
33 minutes

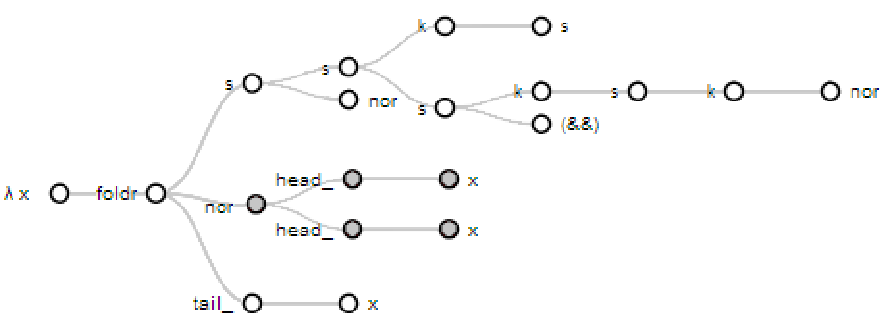
Results



Parent #1, fitness: 0.8125



Parent #2, fitness: 0.5



Child #1, fitness: 1

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

$$=_{\beta\eta}$$

$$\lambda\, x\ .\ foldr\ (\lambda\, y\ z\ .\ nor\ (and\ y\ z)\ (nor\ y\ z))\quad (nor\ (head'\ x)\ (head'\ x))\ \ (tail'\ x)$$

Which is equivalent to:

$$\lambda\, x\ .\ foldr\ xor\ (not\ (head'\ x))\ \ (tail'\ x)$$

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

Results comparison

Conclusions

- How to generate lambda terms in *lnf*
  - 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
roman@cs.cas.cz