Noname manuscript No.
(will be inserted by the editor)

Higher-order Specification of Terminating Functions in Genetic Programming

Omar Montaño-Rivas · César Guerra-García · Moa Johansson · Martín Hernández-Ordoñez

Received: date / Accepted: date

Abstract A fundamental problem in genetic programming is the description of the solution space to restrict the search. This paper present a logical specification of the solution space using higher order formulas. Our technique allows to prove well-definedness properties, such as termination, of synthesized functions using standard facilities in proof assistants. We describe the application of evolving total computable functions in inductive theories. We report the computational effort required to evolve summation over lists, addition and multiplication of natural numbers, odd, even and the Ackermann function. We found that the specification mechanism based in higher-order formulas is flexible enough to represent different kinds of recursion, even general recursion. The techniques described are implemented in the interactive proof assistant Isabelle.

Keywords genetic programming · lambda calculus · recursion · term-rewriting · termination · proof assistant

1 Introduction

Genetic programming (GP) is a machine learning technique used to generate a computer program to solve a specified problem. It uses a stochastic algorithm

Omar Montaño-Rivas and César Guerra-García
Universidad Politécnica de San Luis Potosí, Department of Information Technology,
Urbano Villalón 500, La Ladrillera,78369 San Luis Potosí, S.L.P., México.
E-mail: omar.montano@upslp.edu.mx
E-mail: cesar.guerra@upslp.edu.mx

Moa Johansson
Chalmers University of Technology, Department of Computer Science and Engineering,
MaskingrÃd'nd 2, 412 96 Gothenburg, Sweden.
E-mail: moa.johansson@chalmers.se

Martín Hernández-Ordoñez
Universidad Politécnica de Victoria, Department of Mechatronics,
Carretera Victoria-Soto la Marina Km. 5.5,Parque Científico y Tecnológico de Tamaulipas,
87138 Ciudad Victoria, Tam. México.
E-mail: mhernandezo@upv.edu.mx

inspired by biological evolution to search through a space of possible programs for one that performs "well" according to a fitness function.

A fundamental problem in genetic programming is the description of the solution space to restrict the search. This description needs to be general and flexible enough to represent different kinds of recursion such as primitive recursion, mutual recursion, or even general recursion.

One simple example would be to evolve a program to sum the elements of a list. Here a preliminary step of the GP run is to define the terminal and function set. The set usually found in literature for this problem is the if-then-else construct IF ($\lambda x\, y\, z.\ if\ x\ then\ y\ else\ z$), equality over lists ($=$) and the empty list ($[]$), addition of natural numbers ($+$) and its neutral element (0), the destructor functions *head* and *tail* which separates a lists into its constituent elements, and finally the same function to be defined in order to perform recursion (e.g. f). We can view the search space for this problem as the set of (typed) pure λ-terms N such that equation (1) holds.

$$f\ xs = N\ xs\ IF\ (=)\ []\ (+)\ 0\ head\ tail\ f. \tag{1}$$

Depending on the value of N, the induced definition will be well-defined or not. For example, the λ-term ($\lambda a\, b\, c\, d\, e\, f\, g\, h\, i.\ a$) induces a well-defined function as it yields the identity function $f\ xs = xs$ as opposed to the λ-term ($\lambda a\, b\, c\, d\, e\, f\, g\, h\, i.\ i\ a$) as it yields the non-terminating function $f\ xs = f\ xs$. A λ-term that synthesizes the target function for this problem could be ($\lambda a\, b\, c\, d\, e\, f\, g\, h\, i.\ b\ (c\ d\ a)\ f\ (e\ (g\ a)\ (i\ (h\ a))))$ as it yields, after β-contraction, the following definition.

$$f\ xs = (if\ [] = xs\ then\ 0\ else\ hd\ xs\ +\ f\ (tl\ xs)).$$

In (typed) functional programming languages such as Haskell or SML, it is more common to define recursive functions by pattern matching if the function is being defined recursively on an inductive datatype. For our simple example, Equation 1 can be phrased using pattern matching as

$$\begin{aligned} f\ [] &= 0 \\ f\ (x\#xs) &= M\ x\ xs\ (+)\ f. \end{aligned} \tag{2}$$

Note that we do not need equality on lists or the construct if-then-else because the distinction between empty and non-empty lists is made explicit using the two patterns of definition (2). The destructors *head* and *tail* are also not needed because the constructor $Cons$ ($\#$) is used in the left hand side of the second equation of (2) to separate the head from the tail of the list. Here again, M is a set of λ-terms with adequate type that describe the new search space, but contrary to the previous example, the new search space contains only well-defined functions, i.e. it defines primitive recursive functions. The target function can be synthesized with the λ-term ($\lambda a\, b\, c\, d.\ c\ a\ (d\ b)$) for M. This yields the following definition.

$$\begin{aligned} f\ [] &= 0 \\ f\ (x\#xs) &= x + f\ xs. \end{aligned}$$

Constructor style definitions are usually preferred over destructor style in functional languages because they are more natural and resilient to the problem of partiality of destructive functions, e.g. applying *head* to an empty list would lead to

an exception. Constructor style definitions are also normally shorter, which would imply searching for smaller terms in the search space. Figure 1 shows the number of terms in the search space as a function of the term size for the destructor and constructor styles of definitions (1) and (2) respectively.

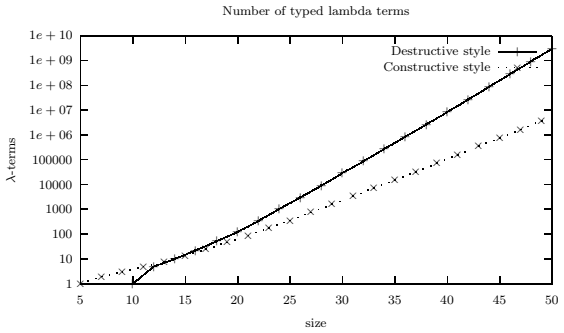


Fig. 1 Logarithmic graph showing the number of λ-terms using the destructor style definition (1) vs the constructor style definition (2).

Mutual recursion is a form of defining functions via recursion involving several functions simultaneously. A standard example of mutual recursion is determining whether a natural number is even or odd by having two separate functions that call each other. A constructor style setting for a GP describing this problem is considering the following definition, where 0 is zero, s is the successor constructor for natural numbers and f and g are the mutually recursive functions to be defined

$$\begin{aligned} f\ 0 &= true \\ f\ (s\ n) &= M\ 0\ s\ n\ g \\ g\ 0 &= false \\ g\ (s\ n) &= N\ 0\ s\ n\ f. \end{aligned} \tag{3}$$

A solution for the target functions is obtained with the following substitution $\{M \mapsto (\lambda a\, b\, c\, d.\ d\ c), N \mapsto (\lambda a\, b\, c\, d.\ d\ c)\}$.

Point-Typing mutation and crossover (see [8]) can be easily adapted to work with our representation of individuals as *substitutions* (see Section 3.3 for a formal description). For example, a *curried* version for *point-typing* mutation is depicted with the individual $\{N \mapsto (\lambda a\, b\, c\, d\, e\, f\, g\, h\, i.\ b\ (c\ d\ a)\ f\ (e\ (g\ a)\ (i\ a)))\}$ from equation (1). The individual is non-terminating as it yields, after β-contraction, the definition

$$f\ xs = (if\ [] = xs\ then\ 0\ else\ hd\ xs\ +\ f\ xs).$$

Mutation can be performed on the individual by exchanging subterm $(i\ a)$ (after β-contraction $f\ xs$) by $(i\ (h\ a))$ (after β-contraction $f\ (tl\ xs)$) synthesizing the target function. An example for a *two-parents* crossover is depicted with the previous individual and $\{N \mapsto (\lambda a\, b\, c\, d\, e\, f\, g\, h\, i.\ b\ (c\ d\ a)\ f\ (e\ (g\ (h\ a))\ (i\ (h\ a))))\}$ (definition $f\ xs = (if\ [] = xs\ then\ 0\ else\ (head\ (tl\ xs)) + f\ (tl\ xs))$) by exchanging the subterm $(i\ a)$ of first individual with $(i\ (h\ a))$ of second individual yielding the target function.

An advantage of using our schematic representation of the problem is that we can use syntactic methods to determine whether two individuals (programs) are equivalent to avoid unnecessary recalculations (see 3.4).

This paper present a formal description of a logical specification of the solution space of functions in a genetic programming algorithm using higher order formulas. This specification allows to synthesize higher-order functions using different recursion schemes (Section 3.1). Well-definedness properties of synthesized functions can be proved by standard facilities, e.g. those provided by an interactive proof assistant such as Isabelle/HOL (Section 3.2). A syntactic method to detect equivalent functions based on normalization is then described in Section 3.4. Section 3.5 describes the overall genetic programming algorithm and the results and discussions over the conducted case studies are reported in Section 4. Preliminary concepts and notations are presented in Section 2.

Although synthesis of algorithms has been studied by different research fields such as inductive programming and deductive program synthesis. In this work we concentrate on genetic programming problems assuming there is a fitness function that is to be minimized/maximized.

2 Background

2.1 Terms of Typed Lambda Calculus

We mainly follow the notation in [11]. Given a finite set $\mathcal{S}$ of *type symbols*, and a denumerable set $\mathcal{S}^\forall$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\forall}$ of *polymorphic types* is generated from these sets by the constructor $\to$ for *functional types*. The type constructor $\to$ associates to the right: read $\tau_1 \to \tau_2 \to \tau_3$ as $\tau_1 \to (\tau_2 \to \tau_3)$. Let τ be a type $\tau_1 \to \ldots \to \tau_n \to \tau_0$ where $\tau_0 \in \mathcal{S}$ and $n \geq 0$. We will sometimes write $\overline{\tau_n} \to \tau_0$ for τ. A *signature* is a set of typed *function symbols* denoted by $\mathcal{F} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^\forall}} \mathcal{F}_\tau$. Terms are generated from a set of typed *variables* $\mathcal{V} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^\forall}} \mathcal{V}_\tau$ and a signature $\mathcal{F}$ by λ-abstraction and application and are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $t : \tau$ to indicate that the term t has type τ.

Typing rules restrict the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ as follows:

$$\frac{x \in \mathcal{V}_\tau}{x : \tau} \qquad \frac{f \in \mathcal{F}_\tau}{f : \tau} \qquad \frac{s : \tau \to \tau' \qquad t : \tau}{(s\ t) : \tau'} \qquad \frac{x : \tau \qquad s : \tau'}{(\lambda x.\, s) : \tau \to \tau'}$$

In the sequel all λ-terms are assumed to be typed.

Instead of $\lambda x_1 \ldots \lambda x_n.\, s$ we also indulge with the notational convenience $\lambda x_1 \ldots x_n.\, s$ or just $\lambda \overline{x_n}.\, s$. Similarly instead of $(\ldots (t\ u_1) \ldots)\, u_n$ we use $t(u_1 \ldots u_n)$ or just $t(\overline{u_n})$.

We differentiate *free variables* from *bound variables* in that the latter are bound by λ-abstraction. The sets of function symbols, free variables and bound variables in a term t are denoted by $\mathcal{F}(t)$, $\mathcal{V}(t)$ and $\mathcal{B}(t)$, respectively. If $\mathcal{V}(M) = \emptyset$, then we say that M is *closed*. We denote $\Lambda^\emptyset := \{M \in \Lambda \mid M \text{ is closed}\}$ as the set of pure closed lambda terms. If $M \in \Lambda^\emptyset(A)$, then we say that M *has type* A or A *is inhabited* by M. The inhabitants of A whose size is smaller than or equal to i is denoted by $\Lambda_i^\emptyset(A) := \{M \in \Lambda_i^\emptyset(A) \mid size(M) \leq i\}$ where the size is defined inductively as:

$$size(M) := \begin{cases} 1 & \text{if } M \text{ is variable,} \\ 1 + size(N) + size(P) & \text{if } M \equiv (N\ P), \\ 1 + size(N) & \text{if } M \equiv (\lambda x.N) \end{cases}$$

We assume the usual definition of α, β and η conversion between λ-terms. We follow the convention that terms which are α-congruent are identified (i.e. $\lambda x.\, x = \lambda y.\, y$). The *$\beta$-normal form* (*$\eta$-normal form*) of a term t is denoted as $t{\downarrow_\beta}$ ($t{\downarrow_\eta}$). Let t be in β-normal form; then t is of the form $\lambda\overline{x_k}.\, a(\overline{u_m})$ where a is called the *head* of t. The *η-expanded* form of t is defined by

$$t{\uparrow^\eta} = \lambda\overline{x_{n+k}}.\, a(\overline{u_m}{\uparrow^\eta}\,, x_{n+1}{\uparrow^\eta}\,, \ldots, x_{n+k}{\uparrow^\eta})$$

where $t : \overline{\tau_{n+k}} \to \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin \mathcal{V}(\overline{u_m})$. We write $t{\updownarrow_\beta^\eta}$ instead of $t{\downarrow_\beta}{\uparrow^\eta}$. We say that a λ-term t is in *long $\beta\eta$-normal form* if $t = t{\updownarrow_\beta^\eta}$.

Example 1 Assume $G : (\tau_1 \to \tau_2) \to \tau_3$, $F : nat \to \tau \to \tau$, $rec : nat \to \tau \to (nat \to \tau \to \tau) \to \tau$, $suc : nat \to nat$, and $c : \tau_1 \to \tau_2$. Some examples of terms and their $\beta\eta$-normal form are shown in the table 1:

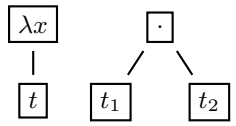
t	$t{\updownarrow_\beta^\eta}$
$\lambda x.\ c\ x$	$\lambda x.\ c\ x$
G	$\lambda x_1 : \tau_1 \to \tau_2.\, G\ (\lambda x_2 : \tau_1.\ x_1\ x_2)$
$G\ c$	$G\ (\lambda x_1 : \tau_1.\ c\ x_1)$
rec	$\lambda x_1, x_2, x_3.\ (rec\ x_1\ x_2\ (\lambda x_4, x_5.\ x_3\ x_4\ x_5))$
F	$\lambda x_1, x_2.\ (F\ x_1\ x_2)$
$rec\ (suc\ x)\ y\ F$	$rec\ (suc\ x)\ y\ (\lambda x_1, x_2.\ F\ x_1\ x_2)$
$F\ x\ (rec\ x\ y\ F)$	$F\ x\ (rec\ x\ y\ (\lambda x_1, x_2.\ F\ x_1\ x_2))$

Table 1: Some higher-order terms and their $\beta\eta$-normal form.

2.2 Substitution and Rewriting

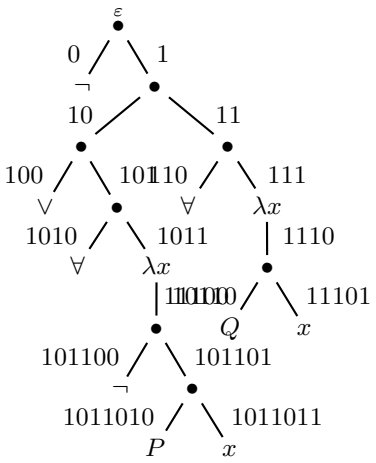
There exist various formalisations of higher-order rewriting. In this manuscript, we will consider higher-order rewrite systems (HRSs) as defined by Nipkow [11].

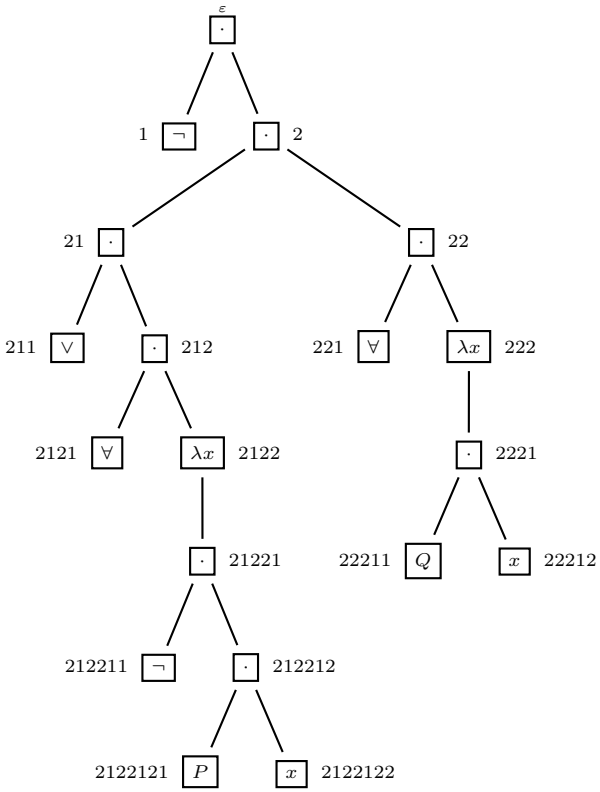
Higher-order terms can be viewed as trees by considering $\lambda x : \sigma.\ \_$ for each variable x and type σ, as a unary function symbol taking the term t as argument to construct the term $\lambda x : \sigma.\, t$. Abstraction and applications yield the following trees:



Positions are strings of positive integers. ε and $\cdot$ denote the empty string (root position) and string concatenation. $\mathcal{P}os(t)$ is the set of positions in t. The *subterm* of t at position p is denoted by $t|_p$. The result of replacing $t|_p$ at position p

in t by u is written $t[u]_p$. For example, the following tree represents the term $\neg((\forall(\lambda x.\neg(P\ x)))\vee(\forall(\lambda x.(Q\ x))))$.





A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a finite mapping from variables into terms of the same type. For $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $Dom(\sigma) = \{x_1, \ldots, x_n\}$

and $Cod(\sigma) = \{t_1, \ldots, t_n\}$. The application of a substitution to a term is defined by

$$\sigma(t) := (\lambda\overline{x_k}.\, t)(\overline{t_n})\downarrow_\beta^\eta\uparrow$$

If there is a substitution σ such that $\sigma(s) = \sigma(t)$ we say s and t are *unifiable*. If there is a substitution σ such that $\sigma(s) = t$ we say that s *matches* the term t. The list of bound variables in a term t at position $p \in \mathcal{P}os(t)$ is denoted as

$$\begin{array}{ll} \mathcal{B}(t,\, \varepsilon) & = [\,] \\ \mathcal{B}((t_1\ t_2),\, i \cdot p) & = \mathcal{B}(t_i,\, p) \\ \mathcal{B}(\lambda x.\, t,\, 1 \cdot p) & = x \cdot \mathcal{B}(t,\, p) \end{array}$$

A pair (l, r) of terms such that $l \notin \mathcal{V}$, l and r are of the same type and $\mathcal{V}(r) \subseteq \mathcal{V}(t)$ is called a *rewrite rule*. We write $l \to r$ for (l, r). A *higher-order rewrite system* (HRS for short) $\mathcal{R}$ is a set of rewrite rules. The rewrite rules of a HRS $\mathcal{R}$ define a reduction relation $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the usual way.

$$s \to_{\mathcal{R}} t \Leftrightarrow \exists (l \to r) \in \mathcal{R},\, p \in \mathcal{P}os(s),\, \sigma.\ \ s|_p = \sigma(l) \wedge t = s[\sigma(r)]_p$$

Example 2 Let

$$HRS = \{\neg(\neg P) \to P,\ \neg(P \vee Q) \to \neg P \wedge \neg Q,\ \neg(\forall(\lambda x.\ P\ x)) \to \exists(\lambda x.\ \neg(P\ x))\}.$$

For readability we use $\forall x.\ P\ x$ and $\exists x.\ P\ x$ instead of $\forall(\lambda x.\ P\ x)$ and $\exists(\lambda x.\ P\ x)$. We also write $\vee$ and $\wedge$ as an infix. Then $\neg((\forall x.\ \neg(P\ x)) \vee (\forall x.\ Q\ x)) \to \neg(\forall x.\ \neg(P\ x)) \wedge \neg(\forall x.\ Q\ x)) \to (\exists x.\ \neg\neg(P\ x)) \wedge \neg(\forall x.\ Q\ x)) \to (\exists x.\ P\ x) \wedge \neg(\forall x.\ Q\ x)) \to (\exists x.\ (P\ x)) \wedge (\exists x.\ \neg(Q\ x))$, where the first reduction takes place at position $p_1 = \varepsilon$ with the second identity and with the substitution $\sigma_1 = \{P \mapsto \forall x.\neg(P\ x),\, Q \mapsto \forall x.Q\ x\}$, the second reduction takes place at position $p_2 = 1 \cdot 2$ with the third identity and with the substitution $\sigma_2 = \{P \mapsto \lambda x.\ \neg(P\ x)\}$, the third reduction takes place at position $p_3 = 1 \cdot 2 \cdot 2 \cdot 1$ with the first identity and with the substitution $\sigma_3 = \{P \mapsto P\ x\}$ and the last reduction takes place at position $p_4 = 2$ with the third identity and with the substitution $\sigma_4 = \{P \mapsto \lambda x.\ Q\ x\}$. The reduction sequence is ilustrated in figure 2.

2.3 Notations

We mainly follow the notation in [13]. A *signature* is a set of typed *function symbols* denoted by $\mathcal{F}$. Terms are generated from a set of typed *variables* $\mathcal{V}$ and a signature $\mathcal{F}$ by λ-abstraction and application and are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

When ambiguities arise, variables inside terms begin with capital letters and lower case letters stand for function symbols. We differentiate *free variables* from *bound variables* in that the latter are bound by λ-abstraction. The sets of function symbols, free variables and bound variables in a term t are denoted by $\mathcal{F}(t)$, $\mathcal{V}(t)$ and $\mathcal{B}(t)$, respectively. We denote the set of pure lambda terms as $\Lambda = \mathcal{T}(\emptyset, \mathcal{V})$. If $\mathcal{V}(M) = \emptyset$, then we say that M is *closed*. We denote $\Lambda^\emptyset := \{M \in \Lambda \mid M \text{ is closed}\}$ as the set of pure closed lambda terms. If $M \in \Lambda^\emptyset(A)$, then we say that M *has type* A or A *is inhabited* by M. The inhabitants of A whose size is smaller than or equal to i is denoted by $\Lambda_i^\emptyset(A) := \{M \in \Lambda_i^\emptyset(A) \mid size(M) \leq i\}$ where the size is defined inductively as:

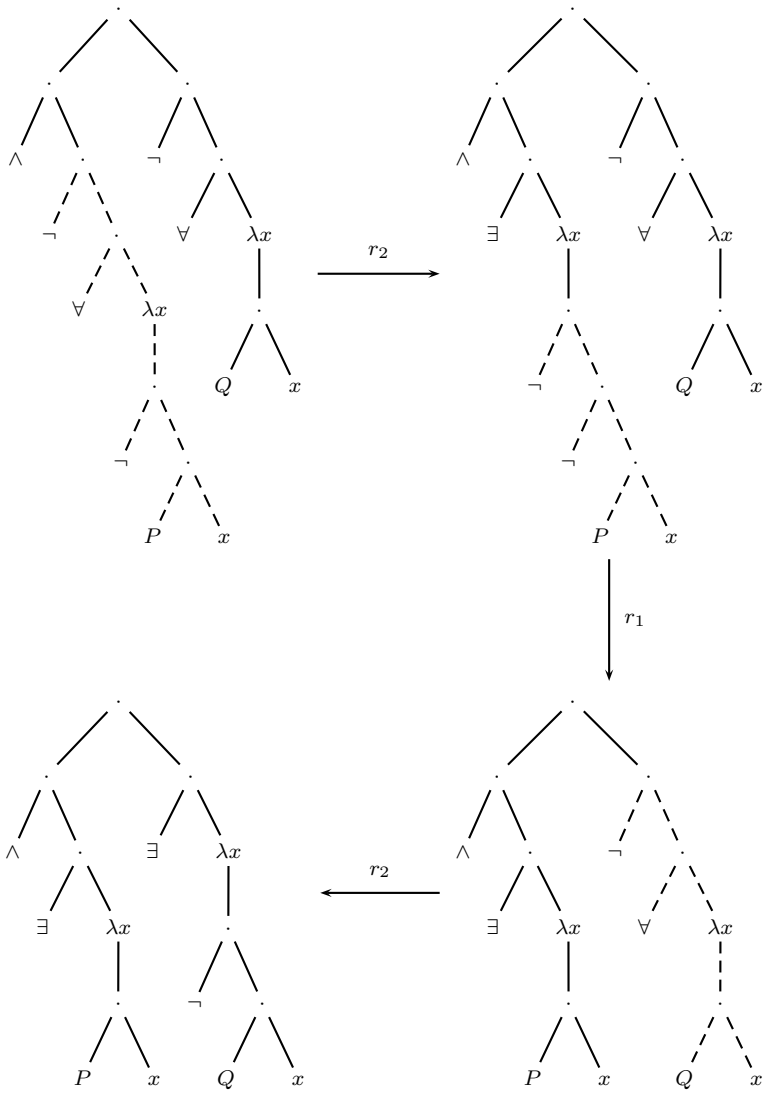


Fig. 2 Reduction sequence over $\neg(\forall x.\, \neg P(x)) \wedge \neg(\forall x.\, Q(x)))$ with the rewrite rules $\neg(\neg P) \to P$ and $\neg(\forall(\lambda x.\, P(x))) \to \forall(\lambda x.\, \neg P(x))$ labeled r_1 and r_2 respectively. The dashed lines indicate the reduced expression within the term (also called redex).

$$size(M) := \begin{cases} 1 & \text{if } M \text{ is variable,} \\ 1 + size(N) + size(P) & \text{if } M \equiv (N\ P), \\ 1 + size(N) & \text{if } M \equiv (\lambda x.N) \end{cases}$$

Positions are strings of positive integers. ε and $\cdot$ denote the empty string (root position) and string concatenation. $\mathcal{P}os(t)$ is the set of positions in term t. The

subterm of t at position p is denoted $t|_p$. The result of replacing $t|_p$ at position p in t by u is written $t[u]_p$.

A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a finite mapping from variables into terms of the same type. For $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $Dom(\sigma) = \{x_1, \ldots, x_n\}$ and $Cod(\sigma) = \{t_1, \ldots, t_n\}$. The application of a substitution to a term is defined by

$$\sigma(t) := (\lambda \overline{x_k}.\, t)(\overline{t_n})\downarrow_\beta^\eta$$

The list of bound variables in a term t at position $p \in \mathcal{P}os(t)$ is denoted as

$$\begin{array}{ll} \mathcal{B}(t,\, \varepsilon) & = [\,] \\ \mathcal{B}((t_1\ t_2),\, i \cdot p) & = \mathcal{B}(t_i,\, p) \\ \mathcal{B}(\lambda x.\, t,\, 1 \cdot p) & = x \cdot \mathcal{B}(t,\, p) \end{array}$$

A pair (l, r) of terms such that $l \notin \mathcal{V}$, l and r are of the same type and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ is called a *rewrite rule*. We write $l \to r$ for (l, r). A *higher-order rewrite system* (HRS for short) $\mathcal{R}$ is a set of rewrite rules. The rewrite rules of a HRS $\mathcal{R}$ define a reduction relation $\to_\mathcal{R}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the usual way.

$$s \to_\mathcal{R} t \Leftrightarrow \exists (l \to r) \in \mathcal{R},\, p \in \mathcal{P}os(s),\, \sigma.\ \ s|_p = \sigma(l) \wedge t = s[\sigma(r)]_p$$

We say that x is *reducible* if there is a y such that $x \to_\mathcal{R} y$. x is in *normal form* (under $\mathcal{R}$) if it is not reducible. The normal form of x is denoted by $x{\downarrow_\mathcal{R}}$. A rewriting system is called *terminating* if there is no sequence of terms $\{a_i | i \in \mathbb{N}\}$ such that $a_i \to a_{i+1}$ for all i.

2.4 Generation of Random Terms

The problem of counting and enumerating β-normal terms of a given type has been studied by a number of researchers [4, 15, 19, 16]. The main idea of the different approaches is to exploit the fixed structure of the terms derived from the type and β-normal shape restrictions to guide the counting and enumeration. In [4] a search algorithm is presented which finds the number of terms in β-normal form given a type. The search algorithm is based on the notion of meta-variables to denote the set of terms of a given type and proceeds by expanding these meta-variables. In [19], a a fixpoint technique is proposed for counting the number of closed terms in η-long β-norm form. The algorithm first constructs a set of polynomial equations extracted from the input type and then solves the set of equations using a least fixpoint algorithm. However, this algorithm diverged when the number of inhabitants of a type is infinite. A grammatical description of the set of β-normal inhabitants of a given type is presented in [15]. This work was latter adapted to produce closed terms in β-normal form of a given size and type uniformly at random assuming a uniform distribution over all terms [16]. In our work, we implemented the ideas described in [16] to ensure two key aspects of the algorithm that choose a member from $x \in \Lambda_i^\emptyset(A)$. Firstly, if A is inhabited, then the algorithm will always return a x, i.e. the algorithm is complete. Second, every x is chosen from $\Lambda_i^\emptyset(A)$ assuming a uniform probability distribution.

3 Theory/calculation

3.1 Synthesis of Functions

In this section we describe the theory used to synthesize functions as a formalization for the material presented in Section 1. Our implementation of the synthesis of functions is based on *schemes* [13]. A scheme is a higher-order formula intended to generate, via substitution, new functions of the underlying theory. However, not every higher-order formula is a scheme. Here, we formally define schemes.

Definition 31 (Schemes)**.** *A* **scheme** $\mathfrak{s}$ *is a (non-recursive) constant definition of a proposition in HOL,* $s_n \equiv \lambda \overline{x}.\, t$*, which we write in the form* $s_n\ \overline{x} \equiv t$*.*

For the scheme $s_n\ \overline{x} \equiv t$, $\overline{x}$ are free variables and $t : bool$ does not contain s_n, does not refer to undefined symbols and does not introduce extra free variables. The scheme (where *dvd* means "divides") $prime\ P \equiv 1 < P \wedge ((dvd\ M\ P) \rightarrow M = 1 \vee M = P)$ is flawed because it introduces the extra free variable M on the right hand side. The correct version is $prime\ P \equiv 1 < P \wedge (\forall m.\ (dvd\ m\ P) \rightarrow m = 1 \vee m = P)$ assuming that all symbols (and types) are properly defined.

Definition 32 (Functional Scheme)**.** *We say that a scheme* $\mathfrak{s} := s_n\ \overline{x} \equiv t$ *is a* **functional scheme** *if* t *has the form* $\exists \overline{f}\ \forall \overline{y}\ \wedge_{i=1}^{m} l_i = r_i$ *and the head of each* $l_i \in \overline{f}$*. The* **defining equations** *of* s *are then denoted by* $l_1 = r_1, \ldots, l_m = r_m$*.*

See [12,13] for examples (and counter-examples) of schemes and functional schemes. In order to describe the technique used to instantiate schemes automatically, here we define some preliminary concepts.

Definition 33 (Schematic Substitutions)**.** *For a scheme* $\mathfrak{s} := u \equiv v$ *with* $\mathcal{V}(s) = \{x_1 : A_1, \ldots, x_n : A_n\}$*, the set of* **schematic substitutions** *with size* i *is defined by:*

$$Sub(\mathfrak{s}, i) := \{\sigma \mid \sigma = \{x_1 \mapsto t_1 \in \Lambda_i^{\emptyset}(A_1), \ldots, x_n \mapsto t_n \in \Lambda_i^{\emptyset}(A_n)\}\}.$$

For a scheme $\mathfrak{s}$, the generated schematic substitutions are used to produce instantiations of s; i.e. functions.

Definition 34 (Instantiation of a Scheme)**.** *Given* $\sigma \in Sub(\mathfrak{s}, i)$*, the* **instantiation of the scheme** $\mathfrak{s} := u \equiv v$ *with* σ *is defined by*

$$Inst(u \equiv v, \sigma) := \sigma(v).$$

We say that an instantiation $i := Inst(\mathfrak{s}, \sigma)$ *is an* **individual** *if there exists a proof of* i*.*

Example 3 Let $\mathcal{F}$ be a signature consisting of $\mathcal{F} = \{0 : nat, s : nat \rightarrow nat\}$. The instantiations induced by the schematic substitutions with size 8 generated from the functional scheme (4) with $P : (nat \rightarrow nat \rightarrow (nat \rightarrow nat \rightarrow nat) \rightarrow nat$ are depicted in table 2.

$$\left(\begin{array}{l} fun\text{-}scheme\ P \equiv \\ \qquad \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (P\ x\ y\ f) \end{array} \right. \end{array} \right) \tag{4}$$

$Sub(s, 8)$	$Inst(s, \sigma_i)$
$\sigma_1 = \{P \mapsto (\lambda x\, y\, z.\, x)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ x \end{cases}$
$\sigma_2 = \{P \mapsto (\lambda x\, y\, z.\, y)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ y \end{cases}$
$\sigma_3 = \{P \mapsto (\lambda x\, y\, z.\, z\ x\ x)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (f\ x\ x) \end{cases}$
$\sigma_4 = \{P \mapsto (\lambda x\, y\, z.\, z\ y\ y)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (f\ y\ y) \end{cases}$
$\sigma_5 = \{P \mapsto (\lambda x\, y\, z.\, z\ x\ y)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (f\ x\ y) \end{cases}$
$\sigma_6 = \{P \mapsto (\lambda x\, y\, z.\, z\ y\ x)\}$	$\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (f\ y\ x) \end{cases}$

Table 2 The instantiations induced by the schematic substitutions with size 8 generated from the functional scheme (4).

Theorem 31 (Proof of well-defined instantiations)**.** *If the defining equations of an instantiation $i := Inst(\mathfrak{s}, \sigma)$ are well-defined, then there exist a proof of i.*

Proof. Since the defining equations induce well-defined functions $\overline{g}$, it is enough to use $\overline{g}$ as witnesses of $\overline{f}$. □

Example 4 Let $\mathcal{F}$ and $\mathfrak{s}$ be the signature and functional scheme of example 3. If $i := Inst(\mathfrak{s}, \sigma_5)$ then the induced defined equations are the following:

$$\begin{aligned} f\ 0\ y &= y \\ f\ (s\ x)\ y &= s\ (f\ x\ y). \end{aligned}$$

It is not difficult to see that these equations are satisfied by the well-known function of addition on the natural numbers $(+)$, i.e. $\forall y.\ 0+y = y$ and $\forall x\ y.\ (s\ x)+ y = s\ (x + y)$. Thus the proof proceeds as follows:

$$\dfrac{\dfrac{\forall y.\ 0 + y = y \qquad \forall x\ y.\ (s\ x) + y = s\ (x + y)}{\forall x\, y.\ \bigwedge \begin{cases} 0 + y = y \\ (s\ x) + y = s\ (x + y) \end{cases}}}{\exists f.\ \forall x\, y.\ \bigwedge \begin{cases} f\ 0\ y\ = y \\ f\ (s\ x)\ y = s\ (f\ x\ y) \end{cases}}$$

3.2 Terminating Individuals

Since we are interested in well-defined functions, well-definedness properties such as existence and uniqueness of the functions generated must be proved. We use Isabelle/HOL's definition facilities (commonly called *function package* [9]) for these proof obligations. The function package allows one to safely define general recursive functions definitions for Isabelle/HOL. The package supports many state of the art features such as partiallity, pattern matching, mutual recursion, among others. It is realised as a definitional extension for Isabelle/HOL following the LCF tradition [2] (recursive definitions are internally transformed into a non-recursive form, such that the function can be defined using standard definition facilities [9].

Starting from the specification of a function the package inductively defines its graph and domain, using the recursive structure of the definition. Then the graph is transformed into a total function modelling the specified function in the domain. The package then proves that the graph actually describes a function on the domain (i.e. function values always exist and are unique). The function package will also automatically derive an induction rule for the definition and prove that the definition is terminating on its domain.

Since our functions are terminating and total by construction, we do not need to handle run-time errors[1]. Most genetic programming systems perform an *ad hoc* handling of run-time errors such as non-termination, but only a few of them guaranties that such run-time errors will never happen.

Although we could prove termination of synthesized functions, in practice, we would still need to stop functions which are expensive to execute. A common practice in the GP community, is to limit the number of recursive calls of the synthesized function by a constant, thus avoiding the function from bringing an entire simulation to a standtill. We can do this by adding an additional counter parameter to each recursive function, i.e. transforming the functional scheme. This approach has another advantage. All functions in the search space, even non-terminating ones, can be forced to terminate. This means that we could prove termination of the transformed functional scheme once and then use it to generate all the functions in the search space, with the guarantee that generated functions will stop after a bounded number of recursive calls. Here we describe how to perform this transformation.

Definition 35 (Terminating closure of a functional scheme)**.** *Given a functional scheme* $\mathfrak{s} := s_n\ \overline{x} \equiv \exists \overline{f}\ \forall \overline{y}\ \wedge_{i=1}^{n} (\wedge_{j=1}^{m} f_i\ \overline{z_j} = r_j)$ *we define the* **terminating closure of the functional scheme** *as:*

$$\widetilde{s_n}\ \overline{x} \equiv \exists \overline{f'}\ \forall \overline{y}\ \overline{y'}\ \wedge_{i=1}^{n} \begin{pmatrix} f_i'\ \overline{x}\ \overline{a_i}\ \overline{v}\ \overline{z} = v_i \quad \wedge \\ \wedge_{j=1}^{m} f_i'\ \overline{x}\ \overline{a}\ \overline{v}\ \overline{z_j} = \sigma(r_j) \end{pmatrix}$$

where $\overline{a} \subset \overline{y'}$ *are the bound variables* $c_1, \ldots, c_n$ *and denote the* **recursion counters** *of the scheme,* $\overline{a_i}$ *are the same as* $\overline{a}$ *but with* $c_i = 0$ *(*$c_1, \ldots, c_{i-1}, 0, c_{i+1}, \ldots, c_n$*),* $\overline{e_i}$ *are the same as* $\overline{a}$ *but with* c_i *decresed by* 1 *(*$c_1, \ldots, c_{i-1}, c_i - 1, c_{i+1}, \ldots, c_n$*),* $\overline{v} \subset \overline{y'}$ *are the values returned by the functions when their number of maximum recursive calls are reached (*$v_1, \ldots, v_n$*),* $\sigma = \{f_1 \mapsto f_1'\ \overline{x}\ \overline{e_1}\ \overline{v}, \ldots, f_n \mapsto f_n'\ \overline{x}\ \overline{e_n}\ \overline{v}\}$ *and* $\overline{z} \subset \overline{y'}$ *are different bound variables.*

Example 5

$$\left(\begin{array}{l} mutual\text{-}scheme\ M\ N \equiv \\ \qquad \exists f\, g.\ \forall x.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ = true \\ f\ (s\ x)\ = M\ x\ 0\ s\ g \\ g\ 0\ = false \\ g\ (s\ x)\ = N\ x\ 0\ s\ f \end{array} \right. \end{array} \right) \tag{5}$$

[1] Although we prove termination of functions, in practice, we still need to stop functions which are expensive to execute.

The terminating closure of the functional scheme (5) is depicted by the functional scheme (6).

$$\left(\begin{array}{l} \widetilde{mutual\text{-}scheme}\ M\ N \equiv \\ \exists f\ g.\ \forall x\ c_f\ c_g\ v_f\ v_g. \\ \quad \bigwedge \left\{\begin{array}{l} f\ M\ N\ 0\ c_g\ v_f\ v_g\ \_\ = v_f \\ f\ M\ N\ c_f\ c_g\ v_f\ v_g\ 0\ = true \\ f\ M\ N\ c_f\ c_g\ v_f\ v_g\ (s\ x)\ = M\ x\ 0\ s\ (g\ M\ N\ c_f\ (c_g - 1)) \\ g\ M\ N\ c_f\ 0\ v_f\ v_g\ \_\ = v_g \\ g\ M\ N\ c_f\ c_g\ v_f\ v_g\ 0\ = false \\ g\ M\ N\ c_f\ c_g\ v_f\ v_g\ (s\ x)\ = N\ x\ 0\ s\ (f\ M\ N\ (c_f - 1)\ c_g) \end{array}\right. \end{array}\right) \tag{6}$$

Theorem 32 (Termination of terminating closures)**.** *Given a functional scheme* $\mathfrak{s} := s_n\ \overline{x}$*, its terminating closure* $\widetilde{s_n}\ \overline{x}$ *only induces terminating individuals.*

Proof. We can prove $\widetilde{s_n}\ \overline{x}$ using Theorem 31. The defining equations induced by $\widetilde{s_n}\ \overline{x}$ are terminating, regardless of the values of $\overline{x}$. This is proved by observing that the summation of the recursion counters is reduced by one in every recursive call. □

Theorem 33 (Size of search space)**.** *The search space induced by a functional scheme* $\mathfrak{s}$ *is the same as its terminating closure* $\widetilde{\mathfrak{s}}$*.*

Proof. Since $\mathcal{V}(\mathfrak{s}) = \{x_1 : A_1, \ldots, x_n : A_n\} = \mathcal{V}(\widetilde{\mathfrak{s}})$, the set of schematic substitutions are the same for both. □

3.3 Crossover and Mutation

In this section we describe two of the most important operators in genetic programming, *crossover* and *mutation*. The simplest of the aforementioned operators is mutation which acts on a single individual at a time. Here we formally define it:

Definition 36 (λ-Mutation)**.** *For a closed term* s *and a size* $i \in \mathbb{N}$ *we denote* $\odot_i\ s$ *as the following set:*

$$\odot_i\ s = \{s[t]_p | \exists p\ t.\ p \in Pos(s) \wedge s|_p : A \wedge t \in \Lambda_i^{\emptyset}(A)\}.$$

A λ-mutation can be extended to a mapping of schematic substitutions to sets of schematic substitutions as follows: for $\sigma := \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ we define

$$\bigodot_i \sigma := \left\{ \rho \,\middle|\, \exists k\ t.\ \begin{array}{c} k \in \{1..n\} \wedge t \in\ \odot_i\ s_k \wedge \\ \rho = \{x_1 \mapsto s_1, \ldots, x_k \mapsto t, \ldots, x_n \mapsto s_n\} \end{array} \right\}.$$

The *crossover* operator works canonically on two individuals. The function of crossover is to exchange genetic material between individuals. Here we formally define the crossover operator in our setting.

Definition 37 (λ-Crossover)**.** *Given two closed terms of the same type* s *and* t*,* $p \in Pos(s)$ *and* $q \in Pos(t)$ *such that* $s|_p$ *and* $t|_q$ *have the same type* τ *and* σ *is a substitution with* $Dom(\sigma) = FV(t|_q)$ *and* $Cod(\sigma) \subseteq BV(s, p)$*, then we denote by* $s\ \overset{\sigma}{p}\otimes_q\ t$ *the term that is obtained from* s *by replacing the subterm at position* p *by* $\sigma(t|_q)$*, i.e.* $s[\sigma(t|_q)]_p$*. We will write* $s \otimes t$ *to denote* $\{w | \exists p\ q\ \sigma\ w.\ w = s\ \overset{\sigma}{p}\otimes_q\ t\}$*.*

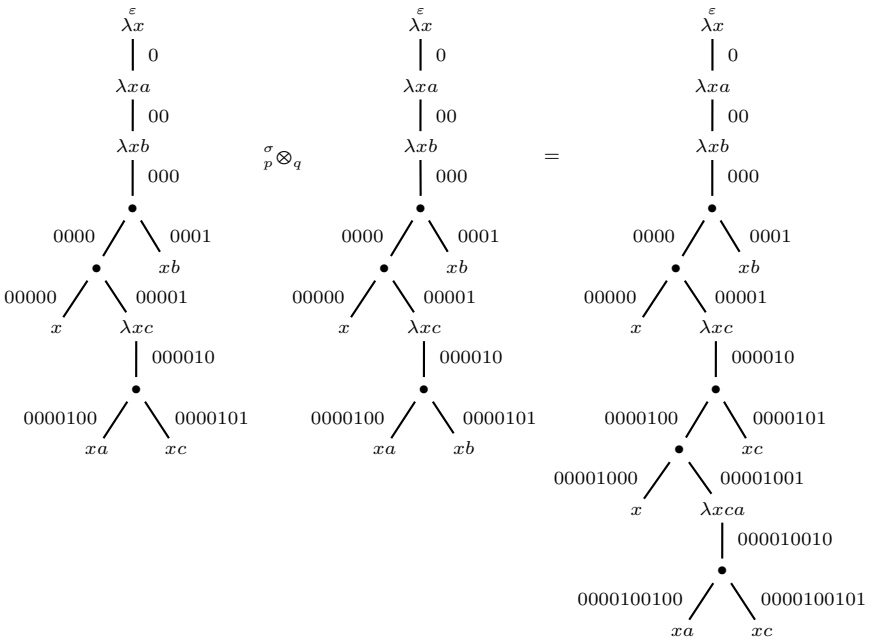


Fig. 3 λ-crossover of two terms where $p = 000010$, $q = 000$ and $\sigma = \{xb \mapsto xc, xa \mapsto xa, x \mapsto x\}$.

Example 6 Figure 3 shows the λ-crossover between two terms s and t with type $((a \to b) \to a \to b) \to (a \to b) \to a \to b$. The crossing point in first term is 000010, in second is 000 and the crossover substitution is $\sigma = \{xb \mapsto xc, xa \mapsto xa, x \mapsto x\}$. Note that $s|_p$ and $t|_q$ have the same type b as it is a requirement for the crossover to be possible.

While it is clear that it is always possible to perform a λ-mutation, it is not so obvious for the λ-crossover case. The Theorem 34 shows that it is always possible to perform the crossover, given two closed terms of the same type.

Theorem 34 (λ-Crossover Existence)**.** *If s and t are two closed terms with the same type, then $\exists p\ q\ \sigma\ x.\ \ {}^{\sigma}_{p}\otimes_{q} = x$.*

Proof. It is sufficient to show that for all closed terms s and t with the same type there exists $p \in Pos(s)$, $q \in Pos(t)$, σ and x such that $x = s\ {}^{\sigma}_{p}\otimes_{q}\ t$. Let $p = q = \epsilon$. Since $FV(t|_\epsilon) = BV(s, \epsilon) = \emptyset$ then we have the identity $I := (\lambda x.\ x)$ as a possible substitution. This yields

$$s\ {}^{I}_{\epsilon}\otimes_{\epsilon}\ t = s[I(t|_\epsilon)]_\epsilon = s[t|_\epsilon]_\epsilon = s[t]_\epsilon = t$$

□

A λ-crossover can be extended to a mapping of pairs of schematic substitutions to sets of schematic substitutions as follows: for $\sigma_1 := \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ and $\sigma_2 := \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define

$$\sigma_1 \bigotimes \sigma_2 := \left\{ \rho \,\middle|\, \exists k\ t.\ \begin{array}{c} k \in \{1 \ldots n\} \wedge t \in s_k \otimes t_k \wedge \\ \rho = \{x_1 \mapsto s_1, \ldots, x_k \mapsto t, \ldots, x_n \mapsto s_n\} \end{array} \right\}.$$

3.4 Syntactic Equivalence of Individuals

Depending on the domain of application, the efficiency of a genetic programming run can be drastically affected by the evaluation of the fitness function. For this reason, Genetic programming systems usually perform some kind of detection of equivalent programs. Here we describe our technique to detect equivalent individuals.

Definition 38 (Equivalence of Individuals)**.** *Given two individuals* $i := Inst(u \equiv v, \sigma)$ *and* $j := Inst(u \equiv v, \tau)$ *and a terminating rewrite system* $\mathcal{R}$*, equivalence of* i *and* j *is denoted as*

$$i \approx_{\mathcal{R}} j := \sigma(v)\big\downarrow_{\mathcal{R}} =_\alpha \tau(v)\big\downarrow_{\mathcal{R}}$$

Normalizing terms by rewriting is a very effective technique used in the automated reasoning field to reduce redundancies. This technique will prevent the re-evaluation of the fitness function of individuals with the same normal form, i.e. equivalent individuals modulo $\mathcal{R}$ are evaluated once.

3.5 Genetic Programming Algorithm

Before we describe the overall procedure, we introduce the auxiliary algorithm `InitialPopulation` to initialize the population. The algorithm receives as arguments a terminating rewrite system $\mathcal{R}$, a terminating closure functional scheme $\widetilde{\mathfrak{s}}$, a $i \in \mathbb{N}$ denoting the maximum size of schematic variables and the population size ps.

```
   InitialPopulation(R, s̃, fitness, i, ps)
1  I := ∅ and T := ∅
2  do
3     σ := {x1 ↦ s1 ∈ Λ_i^∅(A1), ..., xn ↦ sn ∈ Λ_i^∅(An)}
4     ŝ := inst(s̃, σ_j)↓_R
5     if ŝ ∉ T then
6        T[ŝ] := fitness ŝ
7     I := I ∪ {(σ, ŝ, T[ŝ])}
8  while |I| < ps
9  return (I, T)
```

The algorithm iterates in a loop first creating a schematic substitution σ in line 3. Each free variable x_k of $\widetilde{\mathfrak{s}}$ is mapped, using a uniform probability distribution, to a term $s_k \in \Lambda_i^\emptyset(A_k)$ where x_k has type A_k. σ is then used to obtain a normal form w.r.t. $\mathcal{R}$ of the schematic instantiation $Inst(\widetilde{\mathfrak{s}}, \sigma)$ in line 4. This normalised instantiation will prevent the re-evaluation of the fitness function of instantiations with the same normal form with the help of table T, i.e. equivalent instantiations

modulo $\mathcal{R}$ are evaluated only once. Since there is a proof of $\hat{s}$ ($\hat{s}$ is an individual), the fitness function is safely evaluated on $\hat{s}$ in line 6, only if it has not previously been evaluated as checked in line 5. Note that any handling of run-time errors such as non-termination is not needed because well-definedness properties such as termination or totality are guarantied by theorem 32. Then the triplet $(\sigma, \hat{s}, T[\hat{s}])$ is added to the pool I in line 7. Finally, the pool I is returned when the number of individuals is ps.

```
   NewIndividual(R, I, T, s̃, fitness, i, p_mut)
1  select σ_1 and σ_2 from I
2  select σ from σ_1 ⨂ σ_2
3  mutate σ by taking an element from ⨀_i σ
             with probability p_mut
4  ŝ := Inst(s̃, σ)↓_R
5  if ŝ ∉ T then
6    T[ŝ] := fitness ŝ
7  return (T, (σ, ŝ, T[ŝ]))
```

The algorithm `NewIndividual` creates new individuals by performing λ-crossover and λ-mutation. In line 1, two schematic substitutions σ_1 and σ_2 are chosen using a probability directly proportional to the fitness function. The algorithms then performs λ-crossover, λ-mutation and normalization in lines 2, 3 and 4 respectively and an individual is obtained. The fitness function is evaluated on the individual only if it has not previously been evaluated (lines 5 and 6). The algorithm finally returns $(T, (\sigma, \hat{s}, T[\hat{s}]))$ in line 7.

```
   Evolve(R, s̃, fitness, solution, i, ps, p_mut, generations)
1  (I, T) := InitialPopulation(R, s̃, fitness, i, ps)
2  for j := 1 to generations do
3    I' := ∅
4    for k := 1 to ps do
5      (T, (σ, ŝ, f̂)) :=NewIndividual(R, I, T, s̃, fitness, i, p_mut)
6      if solution (σ, ŝ, f̂) then
7        return Some (σ, ŝ, f̂)
8      I' := I' ∪ {(σ, ŝ, T[ŝ])}
9    I := I'
10 return None
```

The overall procedure for genetic programming is described by the pseudocode of `Evolve`. The algorithm receives as arguments a terminating rewrite system $\mathcal{R}$, a terminating closure functional scheme $\widetilde{\mathfrak{s}}$, the fitness function $fitness$, a function $solution$ that tests whether an individual is a solution to the problem at hand, the maximum size of schematic variables $i \in \mathbb{N}$, the population size ps, the probability of mutation p_{mut} and the maximum number of generations the algorithm will run. The algorithm first calls `InitialPopulation` retrieving the initial population I and the table T of precalculated fitnesses in line 1. Then it iterates, first initializing the table I' to $\varnothing$. This table will contain the members of the new generation. Line 4 contains a loop to fill up the table I' by repeatedly calling the `NewIndividual` algorithm on line 5. The returned value is then tested using the function $solution$ and if the individual is a solution then its value is returned by the algorithm. Otherwise it is added to the table I'.

4 Results and discussion

We conducted several case studies in the theory of natural numbers and the theory of lists to synthesize different target functions. We performed 20 independent runs where the control parameters were specified as follows. Population size was set to 500 individuals and the number of generations was fixed to 500. The maximum λ-term size during the construction of the initial population was set to 25 and the same size was used for the λ-mutation operator. We also used *elitism* 5% as a selection scheme. We used two search regimes to search the space of candidate solutions. The first regime used a constructor style functional scheme and the second used a destructor style functional scheme. Supplemental information for the article, such as formal theory documents in human readable Isabelle/Isar notation or the solutions found from the experiments, is presented in url [14].

4.1 Primitive Recursive Functions - summation

The evaluation of the `Evolve` algorithm was performed with a theory consisting of one datatype for lists ([] and #) and addition of natural numbers (+). We use the constructor style functional scheme (7) and the destructor style functional scheme (8) to synthesize summation of natural numbers on a list. We found that the `Evolve` algorithm synthesized the target function in 100% of the cases using both schemes (7) and (8) in 2 or less generations.

$$\left(\begin{array}{l} \textit{constructor-style-sum-scheme } N \equiv \\ \quad \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ [] = 0 \\ f\ (x\#xs) = N\ x\ xs\ (+)\ f \end{array} \right. \end{array}\right) \tag{7}$$

$$\left(\begin{array}{l} \textit{destructor-style-sum-scheme } N \equiv \\ \quad \exists f.\ \forall x\, y.\ f\ xs = N\ xs\ IF\ (=)\ []\ (+)\ 0\ hd\ tl\ f \end{array}\right) \tag{8}$$

The syntactic equivalence of individuals technique detected that 4.2% of the individuals synthesized were equivalent module $\mathcal{R}$ using functional scheme (7) and 0% using scheme (8). We found two equivalent solutions using scheme (7) and two solutions using scheme (8). The equivalence of solutions could have been detected if we would have used the rewrite rule denoting commutativity of addition $x + y = y + x$, which is not included by default in the simpset of Isabelle/HOL. Since the target function was trivially synthesized in the first or second generation, we do not show the best-of-generation individuals graph on Figure 4. For the same reason the cumulative probabilities of success between the schemes (7) and (8) are not depicted in Figure 5.

4.2 Auxiliary functions - multiplication

The evaluation of the `Evolve` algorithm was performed with a theory consisting of one datatype for the naturals (0 and s). We use the constructor style functional scheme (9) and the destructor style functional scheme (10) to synthesize the multiplication of natural numbers.

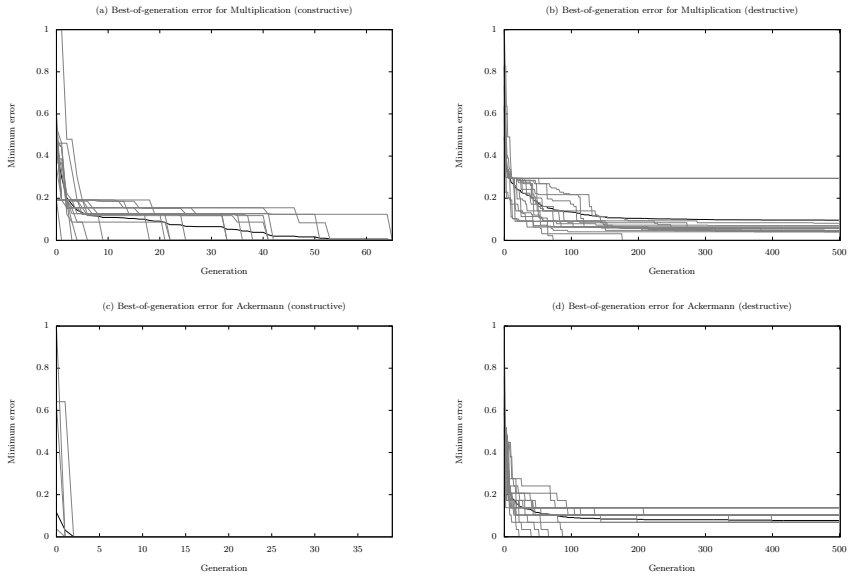


Fig. 4 Best-of-generation error graphs for the two search regimes of case studies.

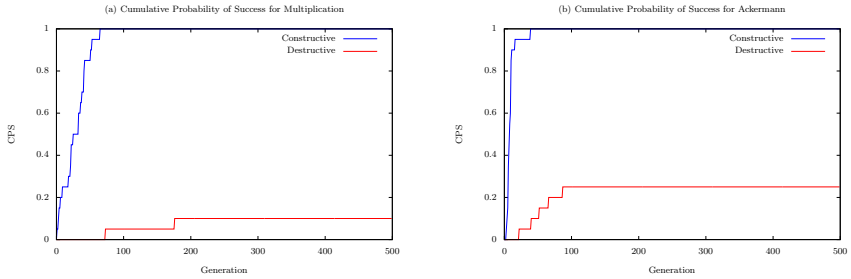


Fig. 5 Comparison of cumulative probability of success between individuals generated for the two search regimes of case studies.

The syntactic equivalence of individuals technique detected that 9.98% of the individuals synthesized were equivalent module $\mathcal{R}$ using functional scheme (9) and 25.4% using scheme (10). We found two equivalent solutions using scheme (9) and two solutions using scheme (10). Figure 4(a) and 4(b) shows the best-of-generation individuals of 20 independent runs using schemes (9) and (10) respectively. Figure 5(a) provides a comparison of the cumulative probabilities of success between the schemes (9) and (10).

$$\left(\begin{array}{l} \textit{constructor-style-mul-scheme}\ M\ N \equiv \\ \qquad \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ y = y \\ f\ (s\ x)\ y = M\ x\ y\ 0\ s\ (f\ x) \\ g\ 0\ y = 0 \\ g\ (s\ x)\ y = N\ x\ y\ 0\ s\ f\ (g\ x) \end{array} \right. \end{array} \right) \tag{9}$$

$$\left(\begin{array}{l} \mathit{destructor\text{-}style\text{-}mul\text{-}scheme}\ M\ N \equiv \\ \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ x\ y = M\ x\ y\ 0\ s\ IF\ (=)\ (f\ x) \\ g\ x\ y = N\ x\ y\ 0\ s\ IF\ (=)\ f\ (g\ x) \end{array} \right. \end{array}\right) \tag{10}$$

4.3 Mutually recursive functions - even/odd

Mutually recursive functions are naturally expressed using our technique. We use the constructor style functional scheme (11) and the destructor style functional scheme (12) to synthesize the even/odd predicates. We found that the `Evolve` algorithm synthesized the target function in 100% of the cases using both schemes (11) and (12) in 15 or less generations. The syntactic equivalence of individuals technique detected that 0% of the individuals synthesized were equivalent module $\mathcal{R}$ using scheme (11) and 2.4% using scheme (12). We only found one solution using scheme (11) and one solution using scheme (12). Since the target funciton was trivially synthesized in the firsts generations, we do not include the best-of-generation individuals graph on Figure 4. We are also not depicting the cummulative probability of success for the same reason.

$$\left(\begin{array}{l} \mathit{constructor\text{-}style\text{-}evenodd\text{-}scheme}\ M\ N \equiv \\ \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ 0 = True \\ f\ (s\ x)\ = M\ x\ 0\ s\ g \\ g\ 0 = False \\ g\ (s\ x)\ = N\ x\ 0\ s\ f \end{array} \right. \end{array}\right) \tag{11}$$

$$\left(\begin{array}{l} \mathit{destructor\text{-}style\text{-}evenodd\text{-}scheme}\ M\ N \equiv \\ \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ x = M\ x\ 0\ True\ False\ (\lambda x.\ x-1)\ IF\ (=)\ g \\ g\ x = N\ x\ 0\ True\ False\ (\lambda x.\ x-1)\ IF\ (=)\ f \end{array} \right. \end{array}\right) \tag{12}$$

4.4 Total computable functions - Ackermann

Total computable functions, such as the Ackerman function, are naturally expressed using our technique. We use the constructor style functional scheme (13) and the destructor style functional scheme (14) to synthesize the Ackerman function. We found that the `Evolve` algorithm synthesized the target function in 100% of the cases using the scheme (13) and 50% of the cases using the scheme (14). The syntactic equivalence of individuals technique detected that 24.85% of the individuals synthesized were equivalent module $\mathcal{R}$ using scheme (13) and 6.62% using scheme (14).

$$\left(\begin{array}{l} \mathit{constructor\text{-}style\text{-}ackermann\text{-}scheme}\ M\ N \equiv \\ \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ y = s\ y \\ f\ (s\ x)\ 0 = M\ x\ 0\ s\ (f\ x) \\ f\ (s\ x)\ (s\ y) = N\ x\ y\ 0\ s\ (f\ x)\ (f\ (s\ x)) \end{array} \right. \end{array}\right) \tag{13}$$

$$\left(\begin{array}{l} \mathit{destructor\text{-}style\text{-}ackermann\text{-}scheme}\ M \equiv \\ \exists f.\ \forall x\, y.\ \bigwedge \left\{ \begin{array}{l} f\ x\ y = M\ x\ y\ 0\ s\ (\lambda x.\ x-1)\ IF\ (=) \\ \qquad\qquad (f\ (x-1))\ (f\ x) \end{array} \right. \end{array}\right) \tag{14}$$

We found three solutions using scheme (13) and five solutions using scheme (14). Figure 4(c) and 4(d) shows the best-of-generation individuals of 20 independent runs using schemes (13) and (14) respectively. Figure 5(b) provides a comparison of the cumulative probabilities of success between the schemes (13) and (14).

4.5 Related Work

Other than this work, the *PolyGP system*[18] and the System F-based GP system [1] are both based on the lambda calculus. The type system ensures that all programs created are type-correct. PolyGP uses a Curry style representation of terms and uses a type unification algorithm to infer types. It is implemented in Haskell and uses different kind of type variables to obtain polymorphism. Redundant expressions in parse trees (introns) are never re-evaluated as Haskell is a non-strict language (uses lazy evaluation). The purity of Haskell, however, has important computational complexity implications during the execution of the system.

The System F-based GP system [1] uses a second order (polymorphic) type system (System F). Since type checking is not decidable in the type system (see [17]), the authors used a Church style representation, where terms were annotated with enough type information. In the GP system, parse trees contain types and types are evolved at the same level as programs (terms). Since System F is strongly normalizable, only total recursive functions are expressible and thus, the primitive recursive functions are contained as a proper subclass.

Probably [5] is the most relevant related work. In the aforementioned work, the author used two control operators to restrict the search space. The composition operator $Cn[f, g_1, \ldots, g_n]$ was defined as the following function application in uncurried form $f(g_1(\overline{x}), \ldots, g_n(\overline{x}))$ and the primitive recursion operator $Pr[f, g]$ was used to define the following primitive recursive functions:

$$\begin{aligned} h(0, \overline{x}) &= f(\overline{x}) \\ h(n+1, \overline{x}) &= g(h(n, \overline{x}), n, \overline{x}). \end{aligned} \tag{15}$$

While the Pr operator is reminiscent to our definition of functional scheme, it has several differences summarized in table 3.

Description	Pr operator	Functional schemes
Application	Curried	Uncurried
Foundation	first-order	higher-order (HOL)
λ-terms	no	yes
Mutual recursion	no	yes
Total recursion	no	yes
Pattern matching	naturals	inductive datatype
User-modifiable scheme	no	yes
Induce terminating functions	yes	yes using its terminating closure
Superpolynomial functions	yes	yes

Table 3 Differences between the Pr operator and functional schemes.

4.6 Limitations and further work

One limitation of our technique is depicted in the following example. Suppose we want to find a λ-term with the type $\alpha \to \beta$ where β is a specialization or an instantiation of α. It is easy to see that no such term exists unless we allow the construction of closed terms, instead of pure λ-terms, with infinitely many type-instantiation functions $\sigma_{\gamma\to\tau} : \mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{F},\mathcal{V})$ where $\sigma_{\gamma\to\tau}$ instantiate type variables from type γ to type τ. In this case, the term $(\lambda x.\, \sigma_{\alpha\to\beta}\ x)$ can be used to embody the term sought. Our technique can be easily extended to avoid this limitation, however we left it as interesting future work.

This work has focused more on genetic programming as an algorithm synthesis technique. However, an interesting further work would be to compare our approach with other other competing techniques to synthesize algorithms not based on genetic programming such as deductive synthesis [10] or inductive synthesis [7,6], e.g. comparing the techniques w.r.t. the dimensions described in [3].

4.7 Conclusions

We have defined a specification mechanism based on higher-order formulas to restrict the solution space of synthesized functions in a GP system. Our technique allows to prove termination of synthesized functions using standard facilities in the interactive proof assistant Isabelle. Our approach to genetic programming was successfully applied to the task of evolving different target functions with different recursion schemes. We found that the specification mechanism proposed is suitable for the tasks to represent different recursion schemes, including general recursion.

Acknowledgments

This work has been supported by Universidad Politécnica de San Luis Potosí, Universidad Politécnica de Victoria and Chalmers University of Technology.

References

1. Binard, F., 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)
2. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF - A mechanised logic of computation, *Lecture Notes in Computer Science*, vol. 78. Springer-Verlag (1979)
3. Gulwani, S.: Dimensions in program synthesis. In: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, pp. 13–24. ACM (2010)
4. Hindley, J.R.: Basic simple type theory. Cambridge tracts in theoretical computer science. Cambridge University Press, New York (1996). URL `http://opac.inria.fr/record=b1090400`
5. Kahrs, S.: Genetic programming with primitive recursion. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pp. 941–942. ACM (2006)
6. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: International Workshop on Approaches and Applications of Inductive Programming, pp. 50–73. Springer (2009)

7. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7**(Feb), 429–454 (2006)
8. Koza, J.R.: Genetic programming ii: Automatic discovery of reusable subprograms. Cambridge, MA, USA (1994)
9. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T. U. München (2009)
10. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems (TOPLAS) **2**(1), 90–121 (1980)
11. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical Computer Science **192**, 3–29 (1998)
12. Montano-Rivas, O., McCasland, R., Dixon, L., Bundy, A.: Scheme-based synthesis of inductive theories. In: Proceedings of the 9th Mexican international conference on Advances in artificial intelligence: Part I, MICAI'10, pp. 348–361. Springer-Verlag, Berlin, Heidelberg (2010). URL `http://dl.acm.org/citation.cfm?id=1927149.1927185`
13. Montano-Rivas, O., McCasland, R., Dixon, L., Bundy, A.: Scheme-based theorem discovery and concept invention. Expert Systems with Applications **39**(2), 1637 – 1646 (2012). DOI 10.1016/j.eswa.2011.06.055. URL `http://www.sciencedirect.com/science/article/pii/S0957417411009559`
14. Montaño Rivas, O.: Higher-order specification of terminating functions in genetic programming - supplementary information (2016). URL `http://atit.upslp.edu.mx/index.php/publications?id=47`. [Online; accessed 14-June-2016]
15. Takahashi, M., Akama, Y., Hirokawa, S.: Normal proofs and their grammar. information and computation **125**(2), 144–153 (1996)
16. Wang, J.: Generating random terms in beta normal form of the simply-typed lambda calculus. Boston University. July **10** (2005)
17. Wells, J.B.: Typability and type checking in system f are equivalent and undecidable. Annals of Pure and Applied Logic **98**(1), 111–156 (1999)
18. Yu, T., Clack, C.: Polygp: A polymorphic genetic programming system in haskell. Genetic Programming **98** (1998)
19. Zaionc, M.: Fixpoint Technique for Counting Terms in Typed Lambda Calculus. Technical report (State University of New York. Department of Computer Science). Department of Computer Science, State University of New York at Buffalo (1995). URL `https://books.google.com/books?id=ActGGwAACAAJ`