Comparing the expressive power of Strongly-Typed and Grammar-Guided Genetic Programming

Alcides Fonseca
Diogo Poças
{alcides,dmpocas}@ciencias.ulisboa.pt
LASIGE, Faculdade de Ciências da Universidade de Lisboa
Lisboa, Portugal

ABSTRACT

Since Genetic Programming (GP) has been proposed, several flavors of GP have arisen, each with their own strengths and limitations. Grammar-Guided and Strongly-Typed GP (GGGP and STGP, respectively) are two popular flavors that have the advantage of allowing the practitioner to impose syntactic and semantic restrictions on the generated programs. GGGP makes use of (traditionally context-free) grammars to restrict the generation of (and the application of genetic operators on) individuals. By guiding this generation according to a grammar, i.e. a set of rules, GGGP improves performance by searching for an good-enough solution on a subset of the search space. This approach has been extended with Attribute Grammars to encode semantic restrictions, while Context-Free Grammars would only encode syntactic restrictions. STGP is also able to restrict the shape of the generated programs using a very simple grammar together with a type system. In this work, we address the question of which approach has more expressive power. We demonstrate that STGP has higher expressive power than Context-Free GGGP and less expressive power than Attribute Grammatical Evolution.

CCS CONCEPTS

• **Theory of computation → Grammars and context-free languages**; • **Mathematics of computing → Genetic programming**.

KEYWORDS

Grammar-Guided Genetic Programming, Strongly-Typed Genetic Programming, Genetic Programming

ACM Reference Format:
Alcides Fonseca and Diogo Poças. 2023. Comparing the expressive power of Strongly-Typed and Grammar-Guided Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590507

GECCO '23, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07…$15.00
https://doi.org/10.1145/3583131.3590507

1 INTRODUCTION

Genetic Programming is a heuristic method for the search problem of finding a program. This abstract method has been successfully applied to different domains, ranging from Program Synthesis [17] to genomics [15]. Genetic Programming is typically used when the search space is so wide that using enumerative methods is not a feasible solution. However, practitioners often have domain knowledge that could further reduce the search space. There are two major approaches for encoding this domain-specific knowledge: (1) consider the constraints in the fitness function, common for soft-constraints; and (2) restrict the representation or genotype-to-phenotype mapping, so invalid individuals are never created in the first place, useful for hard constraints. Grammars and types have been two popular approaches for restricting the program representation [1].

Grammar-Guided GP [34] (GGGP) allowed the practitioner to restrict the language of the program by specifying a context-free grammar. Context-free grammars are used in the generation of individuals, in which the starting non-terminal element of the grammar is recursively expanded until it reaches the maximum depth or a complete program tree. The additional expressive power of grammars allowed GP to be applied to more search-based software engineering domains, like Software Testing [32], Algorithmic Design [5] and Program Synthesis [11]. While the initially proposed methods used a tree-based representation, Grammatical Evolution (GE) [29] is a flavor of GGGP that uses a list-based genotypic representation. We formalize the synthesis procedure of GGGP independently of the representation (thus including both tree-based and GE flavors) in Section 3. Attribute Grammar Evolution [4] (AGE) has been proposed as an extension of Grammatical Evolution that uses the semantic representation of attribute grammars to encode the validation rules of individuals (but not necessarily their semantics). We also formalize this version of GGGP, in Section 6.

Type systems have also been proposed as an alternative to custom grammars, with the same purpose of restricting the generation of program trees. The initial approach, Strongly-Typed GP (STGP) [22], uses simple types like integer and boolean, to guarantee that the generated programs type-checked by construction, thus excluding invalid programs from the search procedure. As an example, STGP has been used to generate unit tests for object-oriented software [33]. STGP has been extended with polymorphic types and higher-order functions [2], approximating the features of mainstream programming languages like Java or C♯. A bidirectional tree generation algorithm that supports polymorphic types has been shown to reduce the search space exponentially [18]. We formalize the synthesis procedure of STGP in Section 4.

GGGP has been consistently presented as being more general than STGP [3, 21, 25, 31], with few disputes of the similarity in expressive power [8, 9], without any evidence. We aim to clarify the issue by comparing the relative expressive power of GGGP, AGE and STGP from a theoretical perspective. An approach with an higher expressive power would be able to restrict the generation of programs more than others. As such, our main results establish a hierarchy between these three models, and we prove that:

- STGP is more expressive than GGGP (Section 5).
- AGE is more expressive than GGGP (Section 7).
- AGE is more expressive than STGP (Section 8).

The contributions of this paper are two-fold. First, we provide practitioners with a ranking in expressive power of the GGGP, STGP and AGE. In our ranking, we evidence examples, which generalize for the limitations in expressing constraints on the search space. Secondly, we lay the theoretical foundation for implementers of other search-space-contraining approaches (e.g., Refined Typed Genetic Programming [8, 9]) to formalize their work and show theoretical properties, especially in comparison to these three baseline approaches. We believe our work serves as a benchmark in theoretical expressive power, much like there are benchmark suites that are used to benchmark empirical performance [12].

2 FORMALIZING PROGRAMS

To evaluate the relative expressive power of the different synthesis approaches, it is necessary to define a common notation for the generated programs. While this representation does not have to reflect the actual syntax tree, or tree representation, in the GP environment, it has to be able to express all possible trees that can be synthesized in any of the approaches.

We define the notion of program tree (PT), which represents a synthesized program without any semantic. Because the semantics is independent of the synthesis procedure, we do not care about the evaluation semantics, which can be Turing or non-Turing complete.

Definition 2.1 (Program tree). Let V be a finite set of non-terminal symbols, and Σ be a finite set of terminal symbols. A program tree over (V, Σ) is either $Leaf(s)$ where $s \in \Sigma$; or $Branch(S, p_1, \ldots, p_n)$ where $S \in V$ and $p_1, \ldots, p_n$ are a finite number of Program Trees (PT). Terminal and non-terminal symbols are disjoint.

We do not enforce an arity n for the non-terminal symbols, i.e. we can use the same symbol with different choices of n. The previous definition also allows for empty branchings, when $n = 0$: if $S \in V$ then $Branch(S)$ is a PT.

The program $(x * y) + 3$ can be represented as `Branch(plus, Branch(times, Leaf(x), Leaf(y)), Leaf(3))`, following a notation very similar to S-expressions in Lisp.

3 FORMALIZING GGGP

Grammar-Guided Genetic Programming (GGGP or G3P) [34] traditionally assumes context-free grammars. In this work, the representation has no consequence, as both Tree-based [35] and Grammatical Evolution [29] approaches restrict trees to follow a CFG, regardless of it happening in the generation of the genotype of individuals, or later in the genotype-to-phenotype mapping. We do acknowledge however, that the exploration of the search space of each representation is not the same [35], but we are concerned only with the expressive power of the method. Christiansen Grammars [26] are a context-dependent alternative, but they are not as popular as context-free grammars in GP, due to their increased complexity. We will cover them later in Section 6.

$$\frac{s \in \Sigma}{(V, \Sigma, R, s) \overset{G}{\rightsquigarrow} Leaf(s)} \quad \text{(CFG-Leaf)}$$

$$\frac{(S, \overline{s_i}) \in R \quad \overline{(V, \Sigma, R, s_i) \overset{G}{\rightsquigarrow} p_i}^{\,i}}{(V, \Sigma, R, S) \overset{G}{\rightsquigarrow} Branch(S, \overline{p_i})} \quad \text{(CFG-Branch)}$$

Figure 1: The Synthesis Rules of GGGP.

3.1 Context-free grammars

Definition 3.1 (Context-free grammar). A Context-free grammar is described by the 4-tuple (V, Σ, R, S) where

(1) V is a finite set of non-terminal symbols.
(2) Σ is a finite set of terminal symbols.
(3) R is a finite relation in $V \times (V \cup \Sigma)^*$.
(4) S is the starting symbol. Unlike in other works, we allow S to be either a member of V or Σ, allowing a CFG to express a language with just one terminal, without using any variables.

We follow the traditional formalization of grammars in Theoretical Computer Science [30], with a small diference: The starting symbol of a grammar is traditionally a non-terminal, whereas we allow terminals. This change is done for convenience of the definition of the synthesis rule (e.g., Figure 1), allowing it to be recursive on the starting symbol (instead of creating an auxiliar function for the first expansion only). This does not change the expressive power of context-free grammars because allowing a terminal s as the starting symbol corresponds to allowing a production $S \rightarrow s$ with starting symbol S.

3.2 Grammar-Guided GP Synthesis

While the original GGGP [34] proposed a direct representation using trees, subsequent approaches have proposed other representations like arrays of integers [29] and arrays of arrays of integers [20]. Our formulation is independent of the representation used, and describes what trees can be generated from a given grammar.

Definition 3.2 (CFG Synthesis). $g \overset{G}{\rightsquigarrow} p$ is a relation between a context-free grammar g and a program tree p that represents that p can be synthesized (obtained) from g.

We make use of inference rules to formalize the synthesis relation, following the extensive work on program synthesis (e.g., Polikarpova and Solar-Lezama [28]) and type systems (e.g., Pierce [27]). While it is not common in the GP community, we found no need to invent a new formalism, when there is one already adopted by a large community, including some work on Typed GP [19]. Briefly, if the premises above the line hold, so does the conclusion below the line. Because branches can have multiple child nodes and non-terminals can expand to several terminals and non-terminals, we use the overline to represent repetition (e.g., $\overline{c}$ represents multiple instances of c.), annotated with the repeating variable if needed,

$$\dfrac{\dfrac{\dfrac{3 \in \Sigma}{(S,3) \in R \quad (V,\Sigma,R,3) \overset{G}{\rightsquigarrow} Leaf(3)}\text{CFG-Leaf}}{(plus, S\ S) \in R \quad (V,\Sigma,R,S) \overset{G}{\rightsquigarrow} Branch(S, Leaf(3))}\text{CFG-Branch} \quad \dfrac{\dfrac{x \in \Sigma}{(S,x) \in R \quad (V,\Sigma,R,x) \overset{G}{\rightsquigarrow} Leaf(x)}\text{CFG-Leaf}}{(V,\Sigma,R,S) \overset{G}{\rightsquigarrow} Branch(S, Leaf(x))}\text{CFG-Branch}}{(S, plus) \in R \quad (V,\Sigma,R,plus) \overset{G}{\rightsquigarrow} Branch(plus, Branch(S, Leaf(3)), Branch(S, Leaf(x)))}\text{CFG-Branch}$$

$$(V,\Sigma,R,S) \overset{G}{\rightsquigarrow} Branch(S, Branch(plus, Branch(S, Leaf(3)), Branch(S, Leaf(x))))$$

Figure 2: Example of the GGGP synthesis of program $3+x$ from a grammar with terminals x, y and integers i, and non-terminals S (start), P (plus) and T (times).

Types	T	::=	$t \mid T \to T$
Typing Contexts	Γ	::=	$\epsilon \mid \Gamma, x : T$
Evaluation Contexts	Δ	::=	$\cdot \mid \Delta, x \mapsto p$
Expressions	e	::=	$x \mid ee \mid \lambda x : T.e$
			$\mid Leaf(n) \mid Branch(n, \overline{e})$

Figure 3: The syntax of STGP programs

also standard in these types of formalizations. Pierce [27] provides a more comprehensive introduction to the notation.

In Figure 2 we show that the example $3 + x$ can be synthesized from a grammar with: terminals x, y and finite number of fixed-width integers i; non-terminals S (start), P (plus) and T (times); rules $S \to P \mid T \mid x \mid y \mid i$, $P \to S\ S$ and $T \to S\ S$. Note that we are not including the plus and times operator on the respective right-side of the production, as they are not required syntactically because they are recorded by the grammar expansion. Figure 2 depicts the proof tree that shows that such an example can be derived from the given grammar.

4 FORMALIZING STGP

In STGP, functions have typed parameters and a return type, which must match. STGP can be applied to any typed language. We chose to use a variation of Simply-Typed Lambda Calculus (STLC) [19] for the sake of simplicity. Multiple arguments can be achieved via currying. We implicitly assume that t ranges over a set of base types, that x ranges over a set of expression variables, and that X ranges over a set of symbols. In STGP there is no distinction between terminal and non-terminal symbols, so we can think of X as ranging over the correspondent of $V \cup \Sigma$ in GGGP.

4.1 Syntax

Using the syntax on Figure 3, our expressions can encode both STLC expressions (using only the three first productions), program trees (using only the last two productions) or a mix of both. We say that expressions that only have variables, abstractions and applications to be in the Expression Normal Form (ENF) and expressions that only have leaves and branches to be in the Tree Normal Form (TNF). Thus, program trees correspond exactly to the expressions in TNF.

4.2 STGP Synthesis

Synthesis in STLC follows a syntax-directed approach (Figure 4), taking inspiration from Synquid [28]. We use $\Gamma \vdash T \overset{S}{\rightsquigarrow} e$ to denote

$$\dfrac{x : T \in \Gamma}{\Gamma \vdash T \overset{S}{\rightsquigarrow} x}\ (\text{STGP-Syn-Var})$$

$$\dfrac{\Gamma \vdash U \to T \overset{S}{\rightsquigarrow} e_1 \quad \Gamma \vdash U \overset{S}{\rightsquigarrow} e_2}{\Gamma \vdash T \overset{S}{\rightsquigarrow} e_1 e_2}\ (\text{STGP-Syn-App})$$

$$\dfrac{fresh(x) \quad \Gamma, x : T \vdash e \overset{S}{\rightsquigarrow} U}{\Gamma \vdash T \to U \overset{S}{\rightsquigarrow} \lambda x : T.e}\ (\text{STGP-Syn-Abs})$$

Figure 4: The Term Synthesis Rules of STGP

$$\dfrac{x \mapsto p \in \Delta}{\Delta \vdash x \longrightarrow p}\ (\text{STGP-Eval-Var})$$

$$\dfrac{\Delta \vdash e_1 \longrightarrow \lambda x : T.p_1 \quad \Delta \vdash e_2 \longrightarrow p_2}{\Delta \vdash e_1 e_2 \longrightarrow p_1[x \mapsto p_2]}\ (\text{STGP-Eval-Beta})$$

Figure 5: The Evaluation Rules of STGP

that expression e can be synthesized from type T under the typing context Γ. Similarly, we use $\Delta \vdash e \longrightarrow p$ to denote that expression e can be evaluated to expression p under the evaluation context Δ, a list of assignments of variables to expressions ($x \mapsto e$). Figures 4 and 5 present the Synthesis and Evaluation rules, using the same notation as before.

Finally, we use $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$ to denote that $\Gamma \vdash T \overset{S}{\rightsquigarrow} e$ and $\Delta \vdash e \longrightarrow p$, for a compatible expression e. For ease of exposition, we introduce the concept of synthesis context to refer to a triple (Γ, Δ, T) where Γ is a typing context, Δ is an evaluation context, and T is a type.

5 COMPARING GGGP WITH STGP

Now that we have defined two models for synthesizing program trees, namely GGGP and STGP, we can compare their corresponding expressive powers. In this section, we show that STGP are strictly more expressive than GGGP.

Theorem 5.1. *For any context-free grammar (V, Σ, R, S), there exists a synthesis context (Γ, Δ, T) such that, for any program tree p,*

$$(V,\Sigma,R,S) \overset{G}{\rightsquigarrow} p \quad \textit{if and only if} \quad \Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p.$$

Proof. Given a context-free grammar (V, Σ, R, S), construct a typing context and evaluation context as follows.

- for every (terminal or non-terminal) symbol X, we introduce a base type X_T;

Terminals:	Zero, One, "foo"
Non-terminals:	Number, Plus, SizeOf, String
Rules:	Number → Zero
	Number → One
	Number → Plus
	Number → SizeOf
	Plus → Number Number
	SizeOf → String
	String → "foo"
Starting Symbol:	Number

Table 1: Grammar example

- let $r = (X, Y_1, \dots Y_n) \in R$ be a production rule; we introduce an expression variable r, and include
  $r : Y_{1T} \to \cdots \to Y_{nT} \to X_T$,
  $r \mapsto \lambda y_1 : Y_{1T}. \cdots \lambda y_n : Y_{nT}.Branch(X, y_1, \cdots, y_n)$
  in the typing and evaluation contexts, respectively. In particular, for $n = 0$ and a production rule $r = (X, \varepsilon)$, we introduce an expression variable r, include $r : X_T$ in the typing context, and include $r \mapsto Branch(X)$ in the evaluation context.
- for any terminal symbol u, we introduce an expression variable u, and include $u : u_T$, $u \mapsto Leaf(u)$ in the typing and evaluation contexts, respectively.

Now let p be a program tree. We show, by induction on p, that $(V, \Sigma, R, S) \overset{G}{\rightsquigarrow} p$ if and only if $\Gamma \vdash S_T \overset{S}{\rightsquigarrow} e$ and $\Delta \vdash e \longrightarrow p$ for some expression e.

First suppose that $p = Leaf(s)$ for some terminal symbol s. If we can derive $(V, \Sigma, R, S) \overset{G}{\rightsquigarrow} p$, by inspection of the rules for CFGSynthesis, we must have used the rule (CFG-Leaf), which implies that $S = s$. Therefore, we can derive $\Gamma \vdash s_T \overset{S}{\rightsquigarrow} s$ using (STGP-Syn-Var) and $\Delta \vdash s \longrightarrow p$ using (STGP-Eval-Var). Conversely, suppose there is an expression e for which we can derive $\Gamma \vdash S_T \overset{S}{\rightsquigarrow} e$ and $\Delta \vdash e \longrightarrow p$. A derivation for $\Delta \vdash e \longrightarrow p$ can only use rules (STGP-Eval-Var) and (STGP-Eval-Beta). The second of these rules creates an expression having p as a subexpression; however, the only axioms (in our evaluation context) containing the $Leaf$ constructor are of the form $u \mapsto Leaf(u)$ with u a terminal symbol. Therefore, we conclude that $e = s$, so that we can derive $\Gamma \vdash S_T \overset{S}{\rightsquigarrow} s$. This then implies that rule (STGP-Syn-Var) was used, and therefore $T = s_T$, so that $S = s$. We can thus conclude that $(V, \Sigma, R, s) \overset{G}{\rightsquigarrow} p$ thanks to rule (CFG-Leaf).

Next, suppose that $p = Branch(X, p_1, \dots, p_n)$ for some non-terminal symbol X and PTs $p_1, \dots, p_n$. If we can derive

$$(V, \Sigma, R, S) \overset{G}{\rightsquigarrow} p,$$

then we must have used the rule (CFG-Branch), which implies that $S = X$, that there exist symbols $s_1, \dots, s_n$ with $r = (X, s_1, \dots, s_n) \in R$, and that $(V, \Sigma, R, s_i) \overset{G}{\rightsquigarrow} p_i$ for $i = 1, \dots, n$. By induction hypothesis there exist expressions $e_1, \dots, e_n$ such that

$$\Gamma \vdash s_{iT} \overset{S}{\rightsquigarrow} e_i \text{ and } \Delta \vdash e_i \longrightarrow p_i$$

Moreover, by construction we have an expression variable r with

$$\Gamma \vdash s_{1T} \to \dots \to s_{nT} \to X_T \overset{S}{\rightsquigarrow} r;$$
$$\Gamma \vdash r \longrightarrow \lambda s_1 : s_{1T}. \cdots \lambda s_n : s_{nT}.Branch(X, s_1, \cdots, s_n).$$

Γ	$z : ZT$
	$o : OT$
	$f : fT$
	$n0 : ZT \to NT$
	$n1 : OT \to NT$
	$np : PT \to NT$
	$ns : SzT \to NT$
	$pl : NT \to NT \to PT$
	$sz : StT \to SzT$
	$st : fT \to StT$
Δ	$z \mapsto Leaf(Zero)$
	$o \mapsto Leaf(One)$
	$f \mapsto Leaf("foo")$
	$n0 \mapsto \lambda x : ZT.Branch(Number, x)$
	$n1 \mapsto \lambda x : OT.Branch(Number, x)$
	$np \mapsto \lambda x : PT.Branch(Number, x)$
	$ns \mapsto \lambda x : StT.Branch(Number, x)$
	$pl \mapsto \lambda x : NT.\lambda y : NT.Branch(Plus, x, y)$
	$sz \mapsto \lambda x : StT.Branch(SizeOf, x)$
	$st \mapsto \lambda x : fT.Branch(String, x)$
T	NT

Table 2: Synthesis context corresponding to the context-free grammar in Table 1.

By a successive application of the rules (STGP-Syn-App) as well as (STGP-Eval-Beta), we then derive that $\Gamma \vdash X_T \overset{S}{\rightsquigarrow} rs_1 \dots s_n$ and $\Delta \vdash rs_1 \dots s_n \longrightarrow p$, respectively.

Conversely, suppose that there exists e and derivations for

$$\Gamma \vdash S_T \overset{S}{\rightsquigarrow} e \text{ and } \Delta \vdash e \longrightarrow p.$$

Since the only axioms in Δ that contain the *Branch* constructor are of the form $r \mapsto \lambda y_1 : y_{1T}. \cdots \lambda y_m : y_{mT}.Branch(X, y_1, \cdots, y_m)$, we conclude that our derivation for $\Delta \vdash e \longrightarrow p$ must have used n applications of the rule (STGP-Eval-Beta), ending with the axiom corresponding to some production rule r. Let $e_1, \dots, e_n$ be the intermediate expressions appearing in this derivation, so that in the root of our derivation tree we have $e = re_1 \cdots e_n$ and moreover we have $\Delta \vdash e_i \longrightarrow p_i$ for $i = 1, \dots, n$. We must have then that $\Gamma \vdash S_T \overset{S}{\rightsquigarrow} re_1 \cdots e_n$. Thus, since $r : y_{1T} \to \cdots \to y_{nT} \to X_t$ is the only instance of r in Γ, we conclude that $X = S$ and $\Gamma \vdash y_{iT} \overset{S}{\rightsquigarrow} e_i$. By induction hypothesis, we determine that $(V, \Sigma, R, y_i) \overset{G}{\rightsquigarrow} p_i$. Then, applying rule (CFG-Branch), we conclude that $(V, \Sigma, R, X) \overset{G}{\rightsquigarrow} p$, as desired. □

To illustrate the above construction, consider the context-free grammar in Table 1 for operations between strings and numbers. A possible tree synthesized by this grammar, corresponding to the program `0 + SizeOf("foo")`, is the tree

```
p = Branch(Number, Branch(Plus,
      Branch(Number, Leaf(Zero)),
         Branch(Number, Branch(SizeOf,
            Branch(String, Leaf("foo"))))))
```

Applying the construction of Theorem 5.1, we get the equivalent synthesis context in Table 2.

Let e be the expression `np (pl (n0 z)(ns (sz (st f))))`. It is straightforward to check that $\Gamma \vdash NT \overset{S}{\rightsquigarrow} e$. The following is a proof for $\Delta \vdash e \longrightarrow p$.

1. $\Delta \vdash f \longrightarrow Leaf(foo)$ (Var)
2. $\Delta \vdash st \longrightarrow \lambda x : fT.Branch(String, x)$ (Var)
3. $\Delta \vdash st\ f \longrightarrow Branch(String, Leaf(foo))$ (Beta 1,2)
4. $\Delta \vdash sz \longrightarrow \lambda x : StT.Branch(SizeOf, x)$ (Var)
5. $\Delta \vdash sz\ (st\ f) \longrightarrow Branch(SizeOf, Branch(String, Leaf(foo)))$ (Beta 3,4)
6. $\Delta \vdash ns \longrightarrow \lambda x : SzT.Branch(Number, x)$ (Var)
7. $\Delta \vdash ns\ (sz\ (st\ f)) \longrightarrow Branch(Number, \cdots)$ (Beta 5,6)
8. $\Delta \vdash z \longrightarrow Leaf(Zero)$ (Var)
9. $\Delta \vdash n0 \longrightarrow \lambda x : ZT.Branch(Number, x)$ (Var)
10. $\Delta \vdash n0\ z \longrightarrow Branch(Number, Leaf(Zero))$ (Beta 8,9)
11. $\Delta \vdash pl \longrightarrow \lambda x : NT.\lambda y : NT.Branch(Plus, x, y)$ (Var)
12. $\Delta \vdash (pl\ (n0\ z)\ (ns\ (sz\ (st\ f))) \longrightarrow Branch(Plus, \cdots)$ (Beta 7,10,11)
13. $\Delta \vdash np \longrightarrow \lambda x : PT.Branch(Number, x)$ (Var)
14. $\Delta \vdash np\ (pl\ (n0\ z)\ (ns\ (sz\ (st\ f)))) \longrightarrow Branch(Number, \cdots)$ (Beta 12,13)

To conclude this section, we provide an example of a synthesis context that cannot be described using context-free grammars. Therefore, the reverse of Theorem 5.1 does not hold.

Γ	$z : zT$
	$n0 : zT \to nT$
	$ns : sT \to nT$
	$s0 : zT \to sT$
	$ss : sT \to sT$
	$p : nT \to pT$
Δ	$z \mapsto Leaf(Zero)$
	$n0 \mapsto \lambda x : zT.Branch(Number, x)$
	$ns \mapsto \lambda x : sT.Branch(Number, x)$
	$s0 \mapsto \lambda x : zT.Branch(Succ, x)$
	$ss \mapsto \lambda x : sT.Branch(Succ, x)$
	$p \mapsto \lambda x : nT.Branch(Pair, x, x)$
T	pT

Table 3: Example of a synthesis context to represent pairs of the same number.

Consider the synthesis context in Table 3 to represent pairs of the same number. For a natural number n, let $\lfloor n \rfloor$ be a shorthand notation for the AST

$$\begin{aligned} \lfloor 0 \rfloor &= Branch(Number, Leaf(Zero)) \\ \lfloor 1 \rfloor &= Branch(Number, Branch(Succ, Leaf(Zero))) \\ \lfloor n \rfloor &= Branch(Number, \\ &\quad \underbrace{Branch(Succ, \cdots Branch(Succ}_{n \text{ times}}, Leaf(Zero)) \cdots)) \end{aligned} \tag{1}$$

One can easily check that a program tree p satisfies $\Gamma, \Delta \vdash nT \overset{SE}{\rightsquigarrow} p$ if and only if $p = \lfloor n \rfloor$ for some natural number n. In the same way, we can check that the only program trees p of type pT synthesizable by the above typed calculus must be of the form $p = Branch(Pair, \lfloor n \rfloor, \lfloor n \rfloor)$. However, we cannot build a GGGP that *only* synthesizes such program trees. Any attempt of doing so would result in a GGGP that also synthesizes program trees $p = Branch(Pair, \lfloor n \rfloor, \lfloor m \rfloor)$ for $n \neq m$.

6 FORMALIZING ATTRIBUTE GRAMMAR EVOLUTION

Attribute grammar evolution (AGE) [4] extends Grammar Evolution with Attribute Grammars, allowing the search to be constrained not only by the syntax of programs, but also their semantics.

Attribute grammars [16] are an extension to context-free grammars that enhances them with two set of attributes: synthesized (obtained from child nodes) and inherited (obtained from parents and siblings).

A context-free grammar can be extended into an attribute grammar [14, 16] by introducing:

- for each non-terminal $X \in V$, a set of inherited attributes for X (INH_X);
- for each non-terminal $X \in V$, a set of synthesized attributes for X (SYN_X);
- a semantics mapping f bound to syntactic rules and attributes.

Terminals:	i, arr, name
Non-terminals:	stmt, expr

Rules:	expr → arr [i] {
	  expr.valid ← $0 \le i \wedge i < arr.size$
	  arr.asizes ← expr.asizes
	  i.asizes ← expr.asizes
	}
	$stmt_0$ → name := new int[i]; $stmt_1$ {
	  $stmt_1$.asizes[name] ← n
	  $stmt_0$.valid ← $stmt_1$.valid
	}
	...

Starting Symbol:	stmt

Table 4: An attribute grammar example for restricting the array indices to valid values for each array.

We assume that inherited and synthesized attributes are disjoint; moreover, every attribute a has an associated range of values T_a (the type T of a). Table 4 shows an example of an excerpt of an attribute grammar that guarantees that array accesses are always valid, with attributes that are synthesized ($expr.valid \leftarrow 0 \le i \wedge i < arr.size$) and others that are inherited ($i.asizes \leftarrow expr.asizes$).

The key element of attribute grammars is the semantics f. If r is a production rule of the form $X_0 \longrightarrow X_1 \cdots X_n$, then semantics are defined for every $a_0 \in SYN_{X_0}$ (denoted by the function $f_{(r,0,a_0)}$ and for every $a_i \in INH_{X_i}$ with $1 \le i \le n$ (denoted by the function $f_{(r,i,a_i)}$).

AGE makes use of attributes to describe the conditions that a phenotype must comply with to be considered semantically valid. The derivation process is cancelled when one of these constraints is violated. To formalize this, we assume that there is at least one

$$\frac{s \in \Sigma}{(V, \Sigma, R, s, f) \overset{AG}{\rightsquigarrow} Leaf(s)[]} \text{(AGE-Leaf)}$$

$$\frac{r = (S, \overline{s_i}) \in R \quad f_{(r,0,\text{valid})} \quad \overline{(V, \Sigma, R, s_i, f) \overset{AG}{\rightsquigarrow} p_i[\overline{\overline{a := f_{(r,i,a)}}^{0<j<i}}^{a \in INH_{s_i}}]}^{i}}{(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} Branch(S, \overline{p_i})[\overline{a := f_{(r,0,a)}}^{a \in SYN_S}]} \text{(AGE-Branch)}$$

Figure 6: The Synthesis Rules of AGE.

synthesized attribute, which we denote by the name valid, such that valid $\in SYN_X$ for every non-terminal X, and $T_{\text{valid}} = \{true, false\}$.

We formalize an attribute grammar as a tuple (V, Σ, R, S, f) where (V, Σ, R, S) is the underlying CFG. We omit SYN_X, INH_X from the definition of the tuple for ease of exposition. AGE synthesis is denoted as $(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} p[a_1 := v_1, \ldots a_n := v_n]$, where v_i is the value of attribute a_i and we present the inference rules in Figure 6. We also use the notation $(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} p[\cdot]$ to mean that (V, Σ, R, S, f) synthesizes p for some choice of attribute values.

As described in Section 2.2 of Cruz et al. [4], we consider only L-attribute grammars, those whose inherited attributes only depend on the parent and left-nodes. This limitation occurs for efficiency as L-attribute Grammars can be parsed in one transversal. This limitation of the synthesis is translated in the AGE-Branch iteration of j that has to be smaller than i, the current production item to be expanded.

7 COMPARING GGGP WITH AGE

Theorem 7.1. *For every context-free grammar (V, Σ, R, S), there exists an attribute grammar (V, Σ, R, S, f) such that, for every AST p,*

$$(V, \Sigma, R, S) \overset{G}{\rightsquigarrow} p \quad \textit{if and only if} \quad (V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} p[\cdot].$$

Proof. We can trivially convert a context-free grammar (V, Σ, R, S) into an attribute grammar (V, Σ, R, S, f) by adding the attribute valid $\in SYN_X$, for every non-terminal X, and defining $f(r, 0, \text{valid}) = true$ everywhere. □

Conversely, we can show that AGE has more expressive power than GGGP, so that the converse of Theorem 7.1 does not hold. The attribute grammar in Table 5 can be used to represent pairs of the same number, and thus coincides with the counter-example from the end of the previous section.

In this example, any AST derived from symbol `Number` must be of the form $\lfloor n \rfloor$ for some natural number n, according to the encoding in (1); moreover, the corresponding attribute `value` equals n. Looking at the definition for the attribute `valid` in the production rule for `Pair`, we can see that the only allowed pairs are for numbers of the same value, i.e. $Branch(Pair, \lfloor n \rfloor, \lfloor m \rfloor)$ with $n = m$.

8 COMPARING STGP WITH AGE

In this section, we compare the STGP and AGE models. Our first example shows that some STGP cannot be expressed as AGE, but only because of a very trivial restriction. Consider the following synthesis context.

Terminals:	Zero
Non-terminals:	Number, Pair, Succ
Attributes:	valid in SYN_Number, T_valid = Bool valid in SYN_Succ, T_valid = Bool valid in SYN_Pair, T_valid = Bool value in SYN_Number, T_value = Nat value in SYN_Succ, T_value = Nat

Rules:	Number → Zero {
	Number.valid ← true
	Number.value ← 0
	}
	Number → Succ {
	Number.valid ← Succ.valid
	Number.value ← Succ.value
	}
	Succ → Zero {
	Number.valid ← true
	Number.value ← 1
	}
	Succ_0 → Succ_1 {
	Succ_0.valid ← true
	Succ_0.value ← Succ_1.value + 1
	}
	Pair → Number_1 Number_2 {
	Pair.valid ←
	(Number_1.value = Number_2.value)
	}

Starting Symbol:	Pair

Table 5: An Attribute Grammar example for Pairs.

Γ	$e1 : yT$ $e2 : yT$
Δ	$e1 \mapsto Branch(Yes, Leaf(Zero))$ $e2 \mapsto Branch(No, Leaf(Zero))$
T	yT

Table 6: Example of a synthesis context (Typing context Γ, evaluation context Δ and target type T) for a simple STGP synthesis that cannot be expressed with AGE.

We can trivially check that the only ASTs synthesizable by this calculus are

$$\Gamma, \Delta \vdash yT \overset{SE}{\rightsquigarrow} Branch(Yes, Leaf(Zero))$$
$$\Gamma, \Delta \vdash yT \overset{SE}{\rightsquigarrow} Branch(No, Leaf(Zero))$$

However, context-free grammars can only synthesize trees with the same top-most constructor; in other words, if $g \overset{G}{\rightsquigarrow} p$ and $g \overset{G}{\rightsquigarrow} p'$, then either p, p' are both of the form $Leaf(s)$ for the same terminal symbol s, or they are both of the form $Branch(S, \ldots)$ for the same non-terminal symbol S. This observation carries over to attribute grammars, so that no AGE could be equivalent to the typed calculus above.

To circumvent this obstacle, we introduce an additional fresh non-terminal symbol S and change our interpretation of equivalence accordingly. Instead of $(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} p[\cdot]$, we consider

$(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} Branch(S, p)[\cdot]$, i.e. we add a branch (with non-terminal S) above the root of p. The main theorem of this section is as follows.

Theorem 8.1. *For any synthesis context (Γ, Δ, T), there exists an attribute grammar (V, Σ, R, S, f) such that, for every program tree p,*

$$\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p \quad \textit{if and only if} \quad (V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} Branch(S, p)[\cdot].$$

In order to prove this theorem, we make use of the following result showing that type-checking is decidable for our typed lambda calculus.

Lemma 8.2. *Given a synthesis context (Γ, Δ, T) and a program tree p, one can decide in polynomial time whether $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.*

Proof. Our algorithm will try to construct an expression e, while looking for a proof for $\Gamma \vdash T \overset{S}{\rightsquigarrow} e$ and $\Delta \vdash e \longrightarrow p$; moreover, it will work recursively on the structure of p. Since any proof for $\Delta \vdash e \longrightarrow p$ can only use rules (STGP-Eval-Var) and (STGP-Eval-Beta), the expression e can only be built using expression variables or applications. Let us consider the leftmost application in e; in other words, let us write

$$e = e_0\ e_1 \cdots e_m,$$

where e_0 is an expression variable and $e_1, \ldots, e_m$ are expressions. Then e_0 must appear as an axiom on both Γ and Δ. By separating the initial segment of abstractions in the axiom in Δ, we can write this axiom in the form

$$\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.e_0',$$

where e_0' is not an abstraction; i.e. it is either a variable x, an application $e_1'\ e_2'$, a leaf construct $Leaf(s)$ for some symbol s, or a branch construct $Branch(X, \overline{e'})$ for some sequence of expressions $\overline{e'}$. In addition, we must have $n \leq m$, otherwise we could not apply the rule (STGP-Eval-Beta) enough times to get rid of all the abstractions.

The main idea of our algorithm is that, without loss, e_0' is either a leaf or a branching; if not, we would have an alternative expression $\tilde{e}$ and alternative proofs for $\Gamma \vdash T \overset{S}{\rightsquigarrow} \tilde{e}$, $\Delta \vdash \tilde{e} \longrightarrow p$ where this is indeed the case. We consider each of the possible cases for e_0'.

- Suppose $\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.x_i$, for some $1 \leq i \leq n$. Then $e_0\ e_1 \cdots e_m$ would Beta-reduce to $\tilde{e} = e_i\ e_{n+1} \cdots e_m$ and we could have used expression $\tilde{e}$ instead of e in our proof of $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.
- Suppose $\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.x$, for some variable x different from any x_i. Then $e_0\ e_1 \cdots e_m$ would Beta-reduce to $\tilde{e} = x\ e_{n+1} \cdots e_m$. In order for $\Delta \vdash e \longrightarrow p$ to hold, x must have been part of our contexts; moreover, we could have used expression $\tilde{e}$ instead of e in our proof of $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.
- Suppose $\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.e_1'\ e_2'$, for some expressions e_1' and e_2'. Then $e_0\ e_1 \cdots e_m$ would Beta-reduce to $\tilde{e} = e_1'[\overline{x_i \mapsto e_i}]\ e_2'[\overline{x_i \mapsto e_i}]\ e_{n+1} \cdots e_m$ and we could have used expression $\tilde{e}$ instead of e in our proof of $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.

Thus, in our algorithm that checks whether $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$, we can begin by looking at all expression variables e_0 for which the corresponding e_0' is either a leaf or a branch construct. Note that this constructor must correspond to the top-most constructor of p. Next, we handle these two cases separately.

- Suppose $p = Leaf(s)$ for some terminal symbol s; then, our algorithm looks in the evaluation context for an axiom of the form $\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.Leaf(s)$. For any such axiom, we need to check: whether there exist expressions e_i for which $\Gamma \vdash T_i \overset{S}{\rightsquigarrow} e_i$, which amounts to proving T_i in a suitable fragment of intuitionistic logic; and whether $e_0 : T_1 \to \cdots T_n \to T$ is in the typing context Γ. Both of these properties can be checked efficiently, and if they hold, then as desired $\Delta \vdash e_0\ e_1 \cdots e_n \longrightarrow Leaf(s)$, so that $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.
- Suppose $p = Branch(X, p_1, \cdots, p_n)$ for some non-terminal symbol X and PTs $p_1, \cdots, p_n$; then, our algorithm looks in the evaluation context for an axiom of the form $\Delta \vdash e_0 \longrightarrow \lambda x_1 : T_1 \cdots \lambda x_n : T_n.Branch(X, e_1', \ldots, e_k')$, for some non-terminal symbol X and expressions $e_1', \ldots, e_k'$. These expressions can be built using variables $x_1, \ldots, x_k$ and using the $Leaf$ and $Branch$ constructors; in particular, they cannot use applications or abstractions, as there is no rule in the STGP evaluation calculus for reducing under a branch construct. Thus, we can write $Branch(X, e_1', \ldots, e_k')$ as a 'virtual' PT where we additionally have variables $x_1, \ldots x_n$ standing for variables.
  Our algorithm then performs the following steps. First, it checks whether the program tree p matches the structure of $Branch(X, e_1', \ldots, e_k')$, while establishing a correspondence between the occurring variables x_i and subtrees of p; in particular, if the same variable x_i appears in two different places, then the corresponding subtrees of p must coincide. Second, for any variable x_i not occurring in $Branch(X, e_1', \ldots, e_k')$, we check whether there exists an expression e_i such that $\Gamma \vdash T_i \overset{S}{\rightsquigarrow} e_i$ (which amounts to proving T_i in a suitable fragment of intuitionistic logic). Third, for any variable x_i which does occur in $Branch(X, e_1, \ldots, e_k)$, let p_i be the corresponding subtree of p. Then, we recursively apply our algorithm to check whether $\Gamma, \Delta \vdash T_i \overset{SE}{\rightsquigarrow} p_i$. If the algorithm succeeds, it also produces a suitable expression e_i for which $\Delta \vdash e_i \longrightarrow p_i$. Then, as desired, we have $\Delta \vdash e_0\ e_1 \cdots e_n \longrightarrow p$ and $\Gamma \vdash T \overset{S}{\rightsquigarrow} e_0\ e_1 \cdots e_n$, so that $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$.

We now argue that the above procedure can be implemented in polynomial time. Checking whether a type T can be proven in an implicational fragment of intuitionistic logic can be done efficiently, and this can be done in a preprocessing stage for each T_i appearing in each expression $e = \lambda x_1 : T_1 \cdots \lambda x_n : T_n.e'$ occuring in Δ where e' is either a leaf or a branch (polynomially many choices). Each of the recursive calls boils down to matching a subtree p' of p against an expression $e = \lambda x_1 : T_1 \cdots \lambda x_n : T_n.e'$ in Δ, where e' is either a leaf or a branch. In a dynamic programming fashion, we can store all such pairs (there are linearly many subtrees and linearly many such expressions, so there are quadratically many pairs). To determine if a pair (e, p') is valid, we must match the structure of e' against the structure of p' (which amounts to traversing e' and p' in parallel), check $\Gamma \vdash T_i \overset{S}{\rightsquigarrow} e_i'$ for x_i not occuring in e' (which can be done via a lookup on the result of the preprocessing stage), and check $\Gamma, \Delta \vdash T_i \overset{SE}{\rightsquigarrow} p_i'$ for x_i occuring in e' (which amounts to checking pairs (e_i', p_i') for p_i' which is a subtree of p'). Each of these checks takes linear time, thus yielding a cubic running time in total. □

Proof of Theorem 8.1. Given a synthesis context (Γ, Δ, T), we define an attribute grammar (V, Σ, R, S, f) as follows. We use as terminal symbols all those symbols s for which $Leaf(s)$ appears in any subexpression of any axiom of Δ. We use as non-terminal symbols all those symbols X for which $Branch(X, \ldots)$ appears in any subexpression of any axiom of Δ. Additionally, we include a fresh symbol S. Any non-terminal symbol X different from S has two synthesized attributes: a boolean attribute X.valid which is always defined as $true$, and a string attribute X.tree. Intuitively, this second attribute will encode the entire subtree starting with X. Furthermore, the starting symbol S will have a boolean attribute S.valid.

Finally, we need to describe the ruleset R and the semantics f. Suppose $Branch(X, e_1, \ldots, e_n)$ appears as a subexpression of some axiom of Δ. Then, we include a rule $X \longrightarrow X_1 \ \cdots X_n$, where $X_1, \ldots, X_n$ range over all possible terminal and non-terminal symbols other than S. Furthermore, we define the synthesized attribute X.tree as (here, + denotes string concatenation)

$$X.\text{tree} = \text{“}Branch(X,\text{”} + s_1 + \text{“,”} + \cdots + \text{“,”} + s_n + \text{“)”}$$

where s_i is either the string X_i.tree if X_i is a non-terminal, or the string “$Leaf(X_i)$” if X_i is a terminal symbol.

To conclude, we add a rule $S \longrightarrow s$ for every terminal symbol s, and define S.valid to be $true$ if $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} Leaf(s)$, and $false$ otherwise. Similarly, we add a rule $S \longrightarrow X$ for every non-terminal symbol X other than S, and define S.valid to be $true$ if $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} X$.tree, and $false$ otherwise. Any of these are decidable predicates due to Lemma 8.2.

In this way, the grammar (V, Σ, R, S) allows for arbitrary ASTs built with the leaf and branch constructs. The role of the semantics f is to filter, out of all the possible ASTs, precisely those which can be derived from the typed calculus. By construction, we have that $\Gamma, \Delta \vdash T \overset{SE}{\rightsquigarrow} p$ if and only if $(V, \Sigma, R, S, f) \overset{AG}{\rightsquigarrow} Branch(S, p)[\cdot]$, as we wanted to prove. □

To conclude this section, we argue that AGE have strictly more expressive power than STGP. The main reason is that simply-typed lambda calculus is known not to be Turing-complete (as opposed to untyped lambda calculus), whereas in AGE we have no such restrictions on the semantics f. Let $f : \mathbb{N} \longrightarrow \{true, false\}$ be any computable predicate on the natural numbers, and consider the corresponding AG ag_f defined in Table 7.

If we ignore the semantics, then the underlying context-free grammar generates exactly the natural numbers n via the encoding $\lfloor n \rfloor$ as in (1). By introducing the predicate f into the semantics, we can only generate those natural numbers which satisfy f. Since f can be an arbitrary decidable predicate, we conclude that there are choices of f such that ag_f cannot be represented by any synthesis context (Γ, Δ, T); otherwise we would be able to represent f in simply-typed lambda calculus. For example, one could take f to be a predicate related to the Ackermann function.

9 RELATED WORK

McKay et al. [21] stated that GGGP could be used to restrict types, being equivalent in that usage to STGP, without further discussion about the potential expressive power. Forstenlechner et al.[10] presented a GGGP approach to generalize the support for types in the grammar, showing that very simple types can be modeled as context-free grammars.

Terminals:	Zero
Non-terminals:	Number, Succ
Attributes:	valid in SYN_Number, T_valid = Bool value in SYN_Succ, T_value = Nat

Rules:	Number → Zero {
	  Number.valid ← $f(0)$
	}
	Number → Succ {
	  Number.valid ← f(Succ.value)
	}
	Succ → Zero {
	  Succ.value ← 1
	}
	Succ_0 → Succ_1 {
	  Succ_0.value ← Succ_1.value + 1
	}

Starting Symbol:	Number

Table 7: The attribute grammar ag_f with predicates over natural numbers.

Ephremidis et al. [7] discuss the complexity of attribute grammars with regards to the complexity of the f semantic function. One relevant example is the fact that attribute grammars have been developed to represent an interpreter and typechecker of supersets of the STLC [6, 13].

Nicolau and Agapitos [24] have presented grammar design guidelines that are problem independent. A similar approach could be taken for STGP, given they have the same expressive power (but possible differences in performance). An automatic reduction technique has been proposed to replace a complex grammar with a simpler (but equivalent) one automatically [23].

10 CONCLUSIONS

We have presented proof that STGP is more expressive than context-free grammar GGGP, and that AGE is more expressive than both approaches. The main advantage of STGP comes from the semantic nature of the function application. As such, GGGP and STGP synthesis are equivalent as long as the underlying programming language has the same semantics, which is true in Turing-complete languages, but not in other, also common, configuration languages. Additionally, AGE is more powerful than both, allowing the constraint to be Turing complete (as the target language can always be Turing complete), thus restricting languages that have constraints on sibling, child or parent nodes. The formalization presented also the foundation for future formalisation of extensions to either method, and the baselines to evaluate against. This can be used, for instance, to compare Christiansen Grammatical Evolution [26] and Refined Typed GP [9].

Finally, with STGP and GGGP having very similar expressive power with target Turing-complete language, we recommend practitioners use the more ergonomic version in their applications (e.g., [8]).

ACKNOWLEDGMENTS

This work was supported by *Fundação para a Ciência e Tecnologia* (FCT) in the LASIGE Research Unit under the ref. UIDB/00408/2020 and UIDP/00408/2020, the CMU–Portugal project CAMELOT (LISBOA-01-0247-FEDER-045915), and the RAP project under the reference (EXPL/CCI-COM/1306/2021).

REFERENCES

[1] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. 2019. A Survey of Genetic Programming and Its Applications. *KSII Trans. Internet Inf. Syst.* 13, 4 (2019), 1765–1794. https://doi.org/10.3837/tiis.2019.04.002

[2] Franck Binard and Amy P. Felty. 2008. Genetic programming with polymorphic types and higher-order functions. In *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, Conor Ryan and Maarten Keijzer (Eds.). ACM, 1187–1194. https://doi.org/10.1145/1389095.1389330

[3] Pablo Ramos Criado. 2017. *New techniques for Grammar Guided Genetic Programming: dealing with large derivation trees and high cardinality terminal symbol sets.* Ph. D. Dissertation. Universidad Politécnica de Madrid.

[4] Marina de la Cruz, Alfonso Ortega de la Puente, and Manuel Alfonseca. 2005. Attribute Grammar Evolution. In *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005, Las Palmas, Canary Islands, Spain, June 15-18, 2005, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 3562)*, José Mira and José R. Álvarez (Eds.). Springer, 182–191. https://doi.org/10.1007/11499305_19

[5] Péricles B. C. de Miranda and Ricardo B. C. Prudêncio. 2017. Generation of Particle Swarm Optimization algorithms: An experimental study using Grammar-Guided Genetic Programming. *Appl. Soft Comput.* 60 (2017), 281–296. https://doi.org/10.1016/j.asoc.2017.06.040

[6] Atze Dijkstra and S. Doaitse Swierstra. 2004. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3622)*, Varmo Vene and Tarmo Uustalu (Eds.). Springer, 1–72. https://doi.org/10.1007/11546382_1

[7] Sophocles Ephremidis, Christos H. Papadimitriou, and Martha Sideri. 1987. Complexity characterizations of attribute grammar languages. In *Proceedings of the Second Annual Conference on Structure in Complexity Theory, Cornell University, Ithaca, New York, USA, June 16-19, 1987*. IEEE Computer Society.

[8] Guilherme Espada, Leon Ingelse, Paulo Canelas, Pedro Barbosa, and Alcides Fonseca. 2022. Data Types as a More Ergonomic Frontend for Grammar-Guided Genetic Programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 86–94. https://doi.org/10.1145/3564719.3568697

[9] Alcides Fonseca, Paulo Santos, and Sara Silva. 2020. The Usability Argument for Refinement Typed Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12270)*, Thomas Bäck, Mike Preuss, André H. Deutz, Hao Wang, Carola Doerr, Michael T. M. Emmerich, and Heike Trautmann (Eds.). Springer, 18–32. https://doi.org/10.1007/978-3-030-58115-2_2

[10] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10196)*, James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). 262–277. https://doi.org/10.1007/978-3-319-55696-3_17

[11] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11101)*, Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and L. Darrell Whitley (Eds.). Springer, 197–208. https://doi.org/10.1007/978-3-319-99253-2_16

[12] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *GECCO '21: Genetic and Evolutionary Computation Conference, Lille, France, July 10-14, 2021*, Francisco Chicano and Krzysztof Krawiec (Eds.). ACM, 785–794. https://doi.org/10.1145/3449639.3459285

[13] Ted Kaminski and Eric Van Wyk. 2011. Integrating Attribute Grammar and Functional Programming Language Features. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6940)*, Anthony M. Sloane and Uwe Aßmann (Eds.). Springer, 263–282. https://doi.org/10.1007/978-3-642-28830-2_15

[14] Sven Karol. 2006. An introduction to attribute grammars. *Department of Computer Science. Technische Universitat Dresden, Germany* (2006).

[15] Mohammad Wahab Khan and Mansaf Alam. 2012. A survey of application: Genomics and genetic programming, a new frontier. *Genomics* 100, 2 (2012), 65–71. https://doi.org/10.1016/j.ygeno.2012.05.014

[16] Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Math. Syst. Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511

[17] Krzysztof Krawiec, Iwo Bladek, and Jerry Swan. 2017. Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, Peter A. N. Bosman (Ed.). ACM, 953–960. https://doi.org/10.1145/3071178.3071224

[18] Tomás Kren, Josef Moudrík, and Roman Neruda. 2017. Combining top-down and bottom-up approaches for automated discovery of typed programs. In *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA, November 27 - Dec. 1, 2017*. IEEE, 1–8. https://doi.org/10.1109/SSCI.2017.8285209

[19] Tomás Kren and Roman Neruda. 2014. Generating lambda term individuals in typed genetic programming using forgetful A_*. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*. IEEE, 1847–1854. https://doi.org/10.1109/CEC.2014.6900547

[20] Nuno Lourenço, Filipe Assunção, Francisco B. Pereira, Ernesto Costa, and Penousal Machado. 2018. Structured Grammatical Evolution: A Dynamic Approach. In *Handbook of Grammatical Evolution*, Conor Ryan, Michael O'Neill, and J. J. Collins (Eds.). Springer, 137–161. https://doi.org/10.1007/978-3-319-78717-6_6

[21] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. 2010. Grammar-based Genetic Programming: a survey. *Genet. Program. Evolvable Mach.* 11, 3-4 (2010), 365–396. https://doi.org/10.1007/s10710-010-9109-y

[22] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evol. Comput.* 3, 2 (1995), 199–230. https://doi.org/10.1162/evco.1995.3.2.199

[23] Miguel Nicolau. 2004. Automatic grammar complexity reduction in grammatical evolution. In *The 3rd Grammatical Evolution Workshop: A workshop of the 2004 Genetic and Evolutionary Computation Conference (GECCO-2004), Seattle, Washington, USA, 26-30 June 2004.*

[24] Miguel Nicolau and Alexandros Agapitos. 2018. Understanding Grammatical Evolution: Grammar Design. In *Handbook of Grammatical Evolution*, Conor Ryan, Michael O'Neill, and J. J. Collins (Eds.). Springer, 23–53. https://doi.org/10.1007/978-3-319-78717-6_2

[25] Adam Nohejl. 2011. Grammar-based genetic programming. (2011).

[26] Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. 2007. Christiansen Grammar Evolution: Grammatical Evolution With Semantics. *IEEE Trans. Evol. Comput.* 11, 1 (2007), 77–90. https://doi.org/10.1109/TEVC.2006.880327

[27] Benjamin C Pierce. 2002. *Types and programming languages.* MIT press.

[28] Nadia Polikarpova and Armando Solar-Lezama. 2015. Program Synthesis from Polymorphic Refinement Types. *CoRR* abs/1510.08419 (2015). arXiv:1510.08419 http://arxiv.org/abs/1510.08419

[29] Conor Ryan, J. J. Collins, and Michael O'Neill. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Genetic Programming, First European Workshop, EuroGP'98, Paris, France, April 14-15, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1391)*, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.). Springer, 83–96. https://doi.org/10.1007/BFb0055930

[30] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

[31] Athanasios Tsakonas. 2006. A comparison of classification accuracy of four genetic programming-evolved intelligent structures. *Inf. Sci.* 176, 6 (2006), 691–724. https://doi.org/10.1016/j.ins.2005.03.012

[32] Silvia Regina Vergilio and Aurora Trinidad Ramirez Pozo. 2006. A Grammar-guided Genetic Programming Framework Configured for Data Mining and Software Testing. *Int. J. Softw. Eng. Knowl. Eng.* 16, 2 (2006), 245–268. https://doi.org/10.1142/S0218194006002781

[33] Stefan Wappler and Joachim Wegener. 2006. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*, Mike Cattolico (Ed.). ACM, 1925–1932. https://doi.org/10.1145/1143997.1144317

[34] Peter A Whigham et al. 1995. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, Vol. 16. 33–41.

[35] Peter A. Whigham, Grant Dick, James MacLaurin, and Caitlin A. Owen. 2015. Examining the "Best of Both Worlds" of Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 1111–1118. https://doi.org/10.1145/2739480.2754784