

Data Types as a More Ergonomic Frontend for Grammar-Guided Genetic Programming

Guilherme Espada, Leon Ingelse, Paulo Canelas, Pedro Barbosa, Alcides Fonseca

LASIGE, Faculdade de Ciências, Universidade de Lisboa

me@alcidesfonseca.com

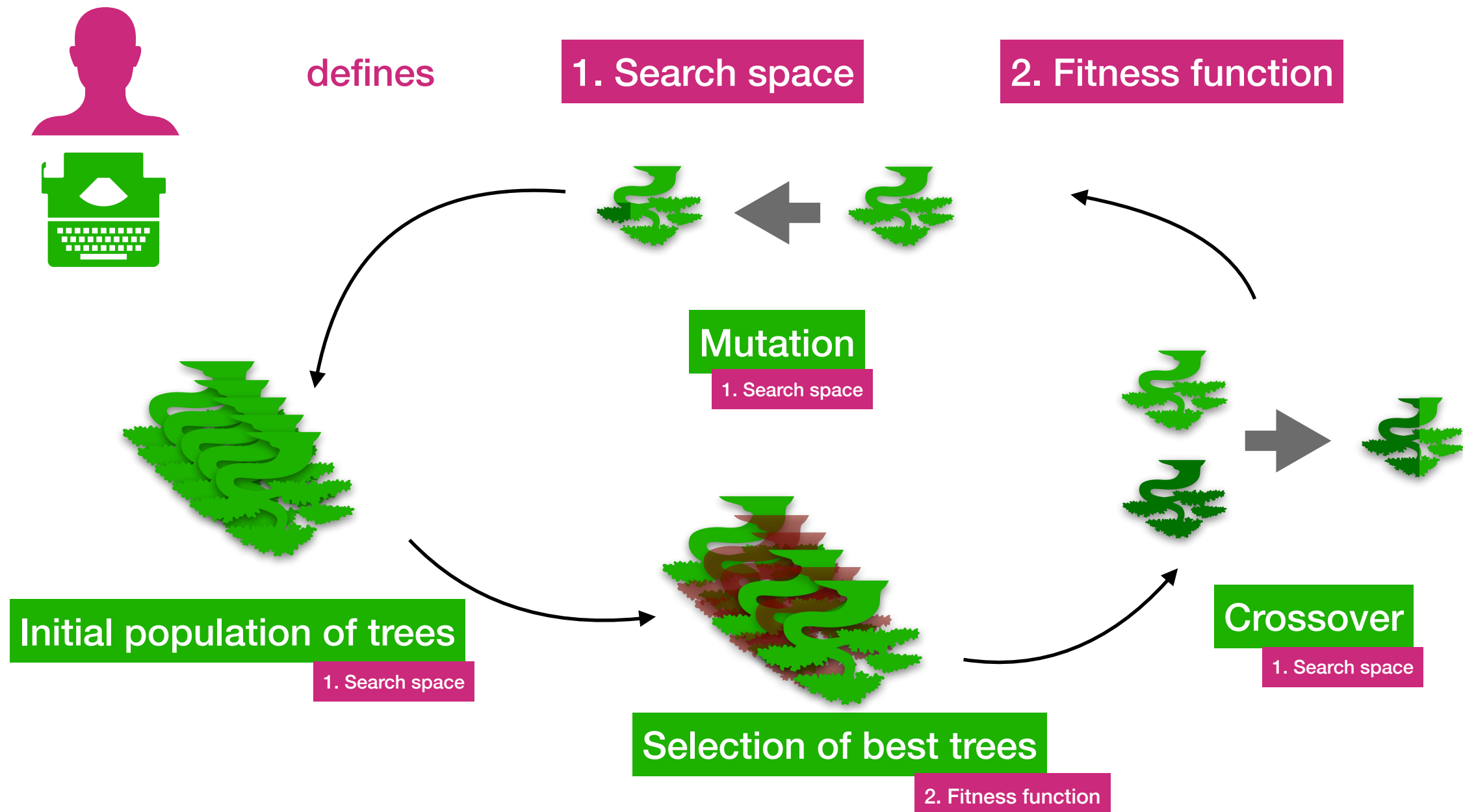
Heuristic method to find good-enough solutions to a complex problem.

- Climate prediction
- Mechatronics
- Diabetes prediction
- Drug discovery
- Financial forecasting
- (many other domains)
- Software fault prediction
- Test case generation
- GPU kernel optimization
- Neural network design
- SuperMario level design

Two different user bases

- Domain Experts
 - Biomedical
 - Physics
 - Economics
 - Often novice programmers
 - Lacking CS concepts
- Software Engineers
 - Advanced Programmers
 - Theory of Computation
 - Programming Languages
 - Compilers

How GP works



DEFINING THE SEARCH SPACE

```
<predefined> ::= 'i0  
'li0 = []; li1 = [];  
'res0 = []'  
<code>  
<code> ::= <code><sta  
<statement> ::= <simp  
<simple_stmt> ::= <ca  
<call> ::= <list_int_  
'<int>')'|'setListInd  
<assign> ::= <int_ass  
<for> ::= 'loopBreak%  
{:\nbreak\n:}loopBrea  
loopBreakConst or sto  
<number> ::= <number>  
<num> ::= '0' | '1' | '2'  
<int_var> ::= 'i0' | 'i  
<int_assign> ::= <int  
<int> ::= <int_var> | '  
<int_arith_ops_protect  
'<int>')'|'abs('<int>  
<arith_ops> ::= '+' | '  
<int_arith_ops_protect  
<arith_prefix> ::= '+  
<list_int_var> ::= 'l  
<list_int> ::= <list_  
<list_int_slice> ::=  
<p> ::= <defp>{:}<callp>  
<defp> ::= def p():{:x = 0.0{:}<code>{:}}return x:  
<callp> ::= XXX_output_XXX = p()  
  
# a function is a list of statements  
<code> ::= <stmt> | <stmt>{:}<code>  
  
# a statement is either an assignment or a for-loop  
<stmt> ::= x = <expr> | <for>  
  
# we don't allow recursive expressions (it becomes easy to hit inf),  
# but we do allow loops, including nested loops.  
<expr> ::= <const> | x | (x + <const>) | (x * <const>)  
<const> ::= 0.5  
  
# we could get nested loops where the inner overwrites the loop  
# variable of the outer, eg:  
  
# for i in range(3):  
#     for i in range(4):  
#         <do stuff>  
  
# but that's ok, because in Py3 scope is done correctly and we are not  
# using the value of i in <do stuff>.  
<for> ::= for i in <seq>{:}<code>{:}  
<seq> ::= [<csitems>] | range(<i>+1)  
<i> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6  
<csitems> ::= <item> | <item>, <csitems>  
<item> ::= <i>
```

- Lack of knowledge and experience in Grammar Design
- Lack of experience with BNF, EBNF and similar languages
- Confusion between the grammar language and the target language
- Dissonance between defining a textual language (stream of characters) and a semantic language (program needs to typecheck)
- In practice: no typechecking, no linting, no static analysis, no autocompletion. Investing in tools is not worthwhile: these languages last only hours in research projects.

Our Approach

Encode Grammars

as Datatypes

- STGP is much more popular in practice than GGGP, due to its simplicity and language integration (DEAP, Jenetics)
- Developers want to use a single language for everything, with consistent semantics and tooling
- STGP and GGGP can be equivalent.

$$A ::= aB \mid c$$
$$B ::= Ad$$

- Each non-terminal is an abstract class
- Each production is a concrete class
- Productions have names
- The focus is on the abstract syntax tree, not the syntax itself.
- Semantics is co-located with the respective syntax

```
class A(ABC): pass
class B(ABC): pass

@dataclass
class A1(A):
    b:B

    def __str__(self):
        return f"a{self.b}"

class A2(A):
    def __str__(self):
        return "c"

@dataclass
class B1(B):
    a:A
    def __str__(self):
        return f"{self.a}d"
```

REAL-WORLD EXAMPLE

```
@dataclass
class Average(BuildingBlock):
    col: Col
    aggregation_col: Annotated[str, VarRange(["target"])]
    window_length: Annotated[int, IntList([10, 25, 50, 75, 100, 150, 200, 300, 400, 600, 800])]

    def assign_target_values_to_data(self, data, historical_data):
        data = data.merge(historical_data, how='left')
        first_data_instant = data[gv.TIME_COLUMN].min()
        historical_data = historical_data[historical_data[gv.TIME_COLUMN] < first_data_instant]

        means = historical_data.groupby(self.col.col_name).mean()[self.aggregation_col]
        uns = data[self.col.col_name].unique()

        def missing_mean_zero(un):
            try:
                return means[un]
            except:
                return 0
        mapping = dict([ (un, missing_mean_zero(un)) for un in uns ])
        def fill_nans(row):
            if np.isnan(row[self.aggregation_col]):
                return mapping[row[self.col.col_name]]
            else:
                return row[self.aggregation_col]
        data[self.aggregation_col] = data.apply(lambda row: fill_nans(row), axis=1)
        # -----
        combined_data = pd.concat([historical_data, data])

        return combined_data

    def aggregate(self, combined_data, instants, window_length):
        aggregated_vals =
combined_data.set_index(gv.TIME_COLUMN).sort_index().groupby(self.col.col_name).rolling(window=window_length,
min_periods=1, closed='left').mean()
        aggregated_vals = aggregated_vals.reset_index().set_index(gv.TIME_COLUMN).sort_index()
        aggregated_vals = aggregated_vals[self.aggregation_col].filter(instants)

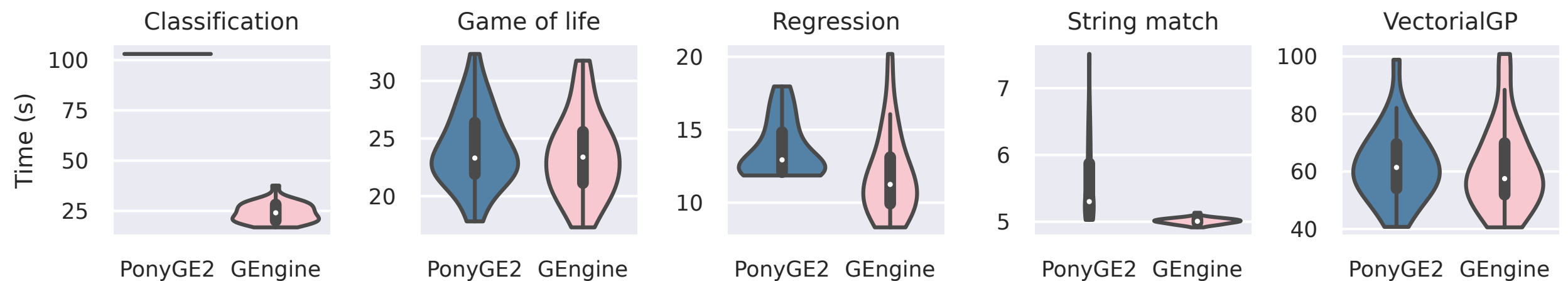
        return aggregated_vals

    def evaluate(self, **kwargs):
        historical_data = kwargs['data']
        historical_data = historical_data[[gv.TIME_COLUMN, self.col.col_name, self.aggregation_col]]
```

```
@weight(3)
@dataclass
class Container(Parent):
    age : Annotated[int, IntRange[0, 200]]
    height : Annotated[float, FloatRange[0.10, 2.5]]
    children : Annotated[List[Container], ListSizeBetween[1,30]]
    var_name : Annotated[str, VarRange["x", "y", "z"]]
```

Evaluation

- BNF is supported, using the encoded presented.
- EBNF features are also supported:
 - Optional symbols can be described using Optional/
Maybe type
 - Lists and repeated symbols can be implemented using
the List type, possible restricted using the
ListBetweenSize metahandler
- Probabilistic Grammars can be encoded using decorators
- Meta-Handlers are equivalent to L-attributed grammars



Performance is on par with the state of the art framework for GGGP (PonyGE2)

Evaluation using Kiper's framework:

	BNF approach	Datatype encoding
Granularity	The BNF syntax may be correct, but not the generated programs.	Type feedback at each level: abstract class, class and semantics
Coherence	Everything is made of strings	Language blocks = classes Semantics = methods Related blocks = modules
Harmony	Mismatch between the BNF language and final elements	Elements of the target language are directly represented

- Our approach is available in Python at
 - <https://github.com/alcides/GeneticEngine>
 - `pip install geneticengine`
- It has been used to improve explainability in:
 - Bike usage prediction
 - Credit classification
 - https://alcidesfonseca.com/research/Tese_Leon.pdf

Data Types as a More Ergonomic Frontend for Grammar-Guided Genetic Programming

Guilherme Espada, Leon Ingelse, Paulo Canelas, Pedro Barbosa, Alcides Fonseca

LASIGE, Faculdade de Ciências, Universidade de Lisboa

me@alcidesfonseca.com

A framework for characterization of the degree of integration of software tools

- Granularity refers to what is the smallest unit which can have feedback, and it should be as fine as possible.
- Coherence refers to how the units are related and integrated with each other.
- Harmony refers to agreement between the elements and the corresponding unit, and should be high (as opposed to dissonant).