






The Usability Argument for Refinement Typed Genetic Programming

Alcides Fonseca^(✉) , Paulo Santos , and Sara Silva 

LASIGE, Faculdade de Ciências da Universidade de Lisboa, Lisbon, Portugal
{alcides,sara}@fc.ul.pt, psantos@lasige.di.fc.ul.pt

Abstract. The performance of Evolutionary Algorithms is frequently hindered by arbitrarily large search spaces. In order to overcome this challenge, domain-specific knowledge is often used to restrict the representation or evaluation of candidate solutions to the problem at hand. Due to the diversity of problems and the unpredictable performance impact, the encoding of domain-specific knowledge is a frequent problem in the implementation of evolutionary algorithms.

We propose the use of Refinement Typed Genetic Programming, an enhanced hybrid of Strongly Typed Genetic Programming (STGP) and Grammar-Guided Genetic Programming (GGGP) that features an advanced type system with polymorphism and dependent and refined types.

We argue that this approach is more usable for describing common problems in machine learning, optimisation and program synthesis, due to the familiarity of the language (when compared to GGGP) and the use of a unifying language to express the representation, the phenotype translation, the evaluation function and the context in which programs are executed.

Keywords: Genetic Programming · Refined types · Search-based software engineering

1 Introduction

Genetic Programming (GP) [28] has been successfully applied in different areas, including bioinformatics [13], quantum computing [35], and supervised machine learning [19]. One of the main challenges of applying GP to real-world problems, such as program synthesis, is the efficient exploration of the vast search space. Frequently, domain knowledge can be used to restrict the search space, making the exploration more efficient. Strongly Typed Genetic Programming (STGP) [24] restricts the search space by ignoring candidates that do not type check. To improve its expressive power, STGP has been extended with type

This work was supported by LASIGE (UIDB/00408/2020) and the CMU—Portugal project CAMELOT (POCI-01-0247-FEDER-045915).

inheritance [10], polymorphism [37] and a Hindley-Milner inspired type system [22], the basis for those in Haskell, SML, OCaml or F \sharp .

Grammar-Guided Genetic Programming (GGGP) [21] also restricts the search space, sometimes enforcing the same rules as STGP, only allowing the generation of individuals that follow a given grammar. Grammar-based approaches have also been developing towards restricting the search space. The initial proposal [21] used context-free grammars (CFG) in the Backus Normal Form. The GAUGE [31] system relies on attribute grammars to restrict the phenotype translation from a sequence of integers. Christiansen grammars [6, 33] can express more restrictions than CFGs, but still have limitations, such as variable scoping, polymorphism or recursive declarations.

We propose Refinement Typed Genetic Programming (RTGP) as a more robust version of STGP through the use of a type system with refinements and dependent types. Languages with these features have gained focus in the Programming Languages (PL) research area: LiquidHaskell [36] is an extension of Haskell that supports refinements; Agda [2] and Idris [3] are dependently-typed languages that are frequently used as theorem provers. These languages support the encoding of specifications within the type system. Previously, special constructs were required to add specification verification within the source code. This idea was introduced in Eiffel [23] and applied later to Java with the JML specification language [18].

In particular, our major contributions are:

- A GP approach that relies on a simple grammar combined with a dependent refined type system, with the argument that this approach is more expressive than existing approaches;
- Concretisation of this approach in the $\mathcal{A}EON$ Programming Language.

These contributions advance GP through a new interface in which to define representations and assess their success. One particular field where our approach might have a direct impact is general program synthesis. We identify two difficulties in the literature [15]: a) the large search space that results from the combination of language operators, grammar and available functions, and b) the lack of a continuous fitness function. We address both aspects within the same programming language.

In the remainder of the current paper we present: the $\mathcal{A}EON$ language for expressing GP problems (Sect. 2); a method for extracting fitness functions from $\mathcal{A}EON$ programs (Sect. 3); the Refined Typed Genetic Programming approach (Sect. 4); examples of RTGP (Sect. 5); a comparison with other approaches, from a usability point of view (Sect. 6); and concluding remarks (Sect. 7).

2 The $\mathcal{A}EON$ Programming Language

We introduce the $\mathcal{A}EON$ programming language as an example of a language with polymorphism and non-liquid refinements. This language can be used as the basis for RTGP due to its support of static verification of polymorphism and

```

type Array<T> { size:Int } // size is a ghost variable

range : (mi:Int, ma:Int) → arr:Array<Int> where (ma > mi and arr.size == ma -
  mi) = native;
append : (a:Array<T>, e:T) → n:Array<T> where (a.size + 1 == n.size) =
  native;
listWith10Elements : (i:Int) → n:Array<Int> where (n.size == 10) {
  append(range(0,i), 42) // Type error
}
fib : (n:Int) → f:Int where (n >= 0 and f >= n) {
  if n < 2 then 1 else fib(n-1) + fib(n-2)
}
incomplete : (n:Int) → r:Int where (r > n && fib(r) % 100 == 0) {
  ■
}

```

Listing 1.1. An example of the $\mathcal{A}EON$ language

a subset of the refinements. However, RTGP is not restricted to this language and could be applied to other languages that have similar type systems.

Listing 1.1 presents a simple example in $\mathcal{A}EON$. To keep $\mathcal{A}EON$ a pure language, several low-level details are implemented in a host language, which $\mathcal{A}EON$ can interact with using the **native** construct. The **range** function is an example of a function whose definition is done in the native language of the interpreter¹.

What distinguishes $\mathcal{A}EON$ from strongly typed mainstream languages like C or Java is that types can have refinements that express restrictions over the types. For instance, the refinements on **range** specify that the second argument must be greater than the first, and the output array has size equal to their different.

The **range** call in the **listWith10Elements** function throws a compile error because i is an Integer, and there are integers that are not greater than 0 (the first argument). The i argument should have been of type $i: \text{Int} \text{ where } i \geq 0$. However, there is another refinement being violated because if $i = 1$, the size of the output will be 2 and not 10 as expected. The correct input type should have been $\{i: \text{Int} \text{ where } i == 9\}$ for the function to compile.

It should now be clear how a language like $\mathcal{A}EON$ can be used to express domain-knowledge in GP problems. A traditional STGP solution would accept any integer value as the argument for **range** and would result in a runtime-error that would be penalized in the fitness function. Individual repair is not trivial to implement without resorting to symbolic execution, which is more computationally intensive than the static verification applied here.

The **incomplete** function, while very basic, is a simple example of the definition of a search problem. The function receives any integer (as there are no restrictions)

¹ We have developed a compiler from $\mathcal{A}EON$ to Java and an $\mathcal{A}EON$ interpreter in Python. In each case, the **range** function would have to be defined in Java and Python.

and returns an integer greater than the one received and whose Fibonacci number is divisible by 100. A placeholder hole (■) is left as the implementation of this function (inspired by Haskell’s and Agda’s `??name` holes [8]). The placeholder allows the program to parse, typecheck, but not execute². Typechecking is required to describe the search problem: Acceptable solutions are those that inhabit the type of the hole, `{r:Int where r > n and fib(r)% 100 == 0}`. This is an example of a dependent refined type as the type of r depends on the value of n in the context. This approach of allowing the user to define a structure and let the search fill in the details has been used with success in sketch and SMT-based approaches [34].

While the $\mathcal{A}EON$ language does not make any distinction, there are two classes of refinements for the purpose of RTGP: liquid and non-liquid refinements. Liquid refinements are those whose satisfiability can be statically verified, usually through the means of an SMT solver. One such example is `{x:Integer where x.size % 2 == 0}` SMT solvers can solve this kind of linear arithmetic problems. Another example is `{x:Array<Integer> where x.size > 0}` because `x.size` is the same as `size(x)` where `size` is an uninterpreted function in SMT solving.

Non-liquid refinements are those that SMT solvers are not able to reason about. These are typically not allowed in languages like LiquidHaskell [36]. One example is the second refinement of incomplete function, `fib(r)% 100 == 0` because the verification of correctness requires the execution of the `fib` function, which can only be called during runtime [7], typically for runtime verification. Another example of a non-liquid refinement would be the use of any natively defined function because the SMT cannot be sure of its behaviour other than the liquid refinement expressed in its type. For instance, when considering a native function that makes an HTTP request, an SMT solver cannot guess statically what kind of reply the server would send.

3 Refinements in GP

Now that we have addressed the difference between liquid and non-liquid refined types, we will see how both are used in the RTGP process. Figure 1 presents an overview of the data flow of the evolutionary process, starting from the problem formulation in $\mathcal{A}EON$ and ending in the solution found, also in $\mathcal{A}EON$. The architecture identifies compiler components and the $\mathcal{A}EON$ code that is either generated or manipulated by those components.

3.1 Liquid Refinements for Constraining the Search Space

Liquid Refinements, the ones supported by Liquid Types [30], are conjunctions of statically verifiable logical predicates of data. We define all other refinements

² Replacing the hole by a crash-inducing expression allows the program to compile or be interpreted. While this is out of scope, the reader may find more in [25].

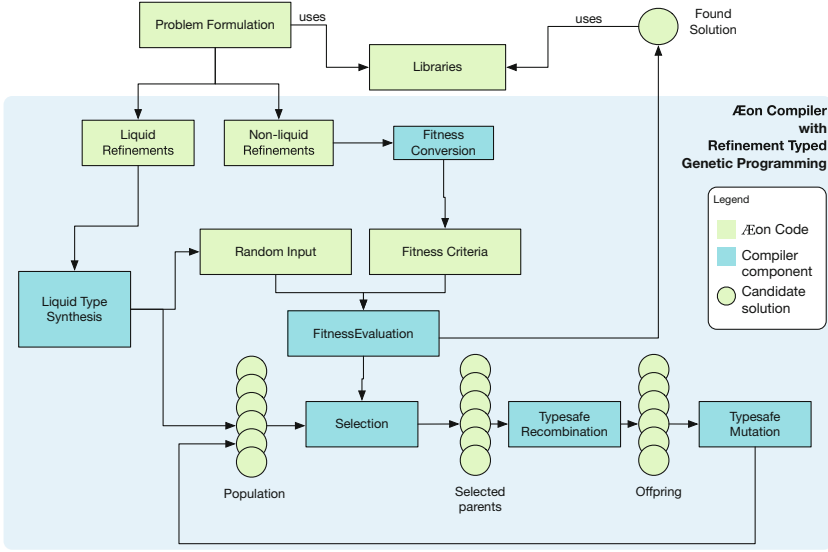


Fig. 1. Architecture of the proposed approach.

as non-liquid refinements. An example of a liquid type is $\{x:\text{Int} \text{ where } x > 3 \text{ and } x < 7\}$, where $x > 3$ and $x < 7$ are liquid refinements.

In our approach, liquid refinements are used to constraint the generation of candidate programs. Through the usage of a type-checker that supports Liquid Types, we are preventing candidates that are known to be invalid from being generated in the first place. However, the use of a type-checker is not ideal, as several invalid candidates might be generated before one that meets the liquid refinement is found. Synquid [29] is a first step in improving the performance of liquid type synthesis. Synquid uses an enumerative approach and an SMT-solver to synthesize programs from a Liquid Type. However, Synquid has several limitations for this purpose: it is unable to synthesize simple refinements related to numerical values (such as the previous example), it is deterministic since it uses an enumerative approach, and while it is presented as complete with regards to the semantic values (can generate all programs that can be expressed in the subset of supported Liquid Types), it is not complete with regards to the syntactic expression. As an example, Synquid is able to synthesize 2 as an integer, but not $1+1$, since the semantic values are equivalent. For the purposes of GP, not being able to synthesize more complex representations of the same program prevents recombination and mutation from exploring small alternatives. We are in the process of formalizing an algorithm that is more complete than Synquid for this purpose.

The RTGP algorithm (Sect. 4) uses the liquid type synthesis algorithm for:

- Generating random input arguments in fitness evaluation (Sect. 3.2);
- Generating a random individual in the initial population;

Table 1. Conversion function f between boolean expressions and continuous values.

Boolean	Continuous		<i>(continued)</i>
true, false	0.0, 1.0	$a \rightarrow b$	$f(\neg a \vee b)$
$x = y$	$\text{norm}(x - y)$	$\neg a$	$1 - f(a)$
$x \neq y$	$1 - f(x == y)$	$x \leq y$	$\text{norm}((x - y))$
$a \wedge b$	$(f(a) + f(b))/2$	$x < y$	$\text{norm}((x - y + \delta))$
$a \vee b$	$\min(f(a), f(b))$		

- Generating a subtree in the mutation operator;
- Generating a subtree in the recombination operator, when the other parent does not have a compatible node.

3.2 Non-liquid Refinements to Express Fitness Functions

A good fitness function is ideally continuous and should be able to measure how close to the real solution a potential solution is. However, fulfilling a given specification is a boolean criterion: it either is fulfilled or not. While previous work [15] has used the number of passed tests as the fitness function, we aim to have a more fine-grained measurement of how far each test is from passing. In particular, we consider the overall error as the fitness value, and the search as a minimization problem.

We propose the use of non-liquid refinement types to synthesize a continuous fitness criteria from the specification (depicted in Fig. 1) that, together with randomly generated input values, is used to obtain the fitness function.

```
f : (n:{Int | n > 0 }, a:{Array<String> | a.size == 3}) → r:Int where (r > n &&
  fib(r) % 100 == 0 and (n > 4 → serverCheck(r) == 3) ) { ■ }
```

Listing 1.2. An example of a specification that corresponds to a bi-objective problem

Listing 1.2 shows an example with a liquid refinement ($r > n$) and two non-liquid clauses in the refinement. Each of these two clauses is handled individually as in a multi-objective problem. Each clause is first reduced to the conjunctive normal form (CNF) and then converted from a predicate into a continuous function that, given the input and expected output, returns a floating point number between 0.0 and 1.0, where the value represents the error. For instance, the example in Listing 1.2 is converted to two functions:

$$\text{norm}(\text{fib}(r)\% 100 - 0) \text{ and } \min(1 - \text{norm}(4 - n + \delta), \text{norm}(\text{serverCheck}(r) - 3))$$

Table 1 shows the conversion rules between boolean expressions and corresponding continuous values. The function f , which is defined using these rules, is applied recursively until a final continuous expression is generated. This approach is an extension of that presented in [12].

Since the output of f is an error, the value **true** is converted to 0.0, stating that condition holds, otherwise 1.0, this being the maximum value of not complying with the condition. Variables and function calls are also converted to 0.0 and 1.0 on whether the condition holds or not. Equalities of numeric values are converted into the normalized absolute difference between the arguments. The normalization is required as it allows different clauses to have the same importance on the given specification. Inequalities are converted to equalities and its difference with 1, negating the fitness result from equality. Conjunctions are converted to the average of the sum of the fitness extraction of both operands. Disjunctions value is obtained by extracting the minimum fitness value of both clauses. The minimum value indicates what clause is the closest to no error. Conditional statements fitness is recursively extracted by using the material implication rule. Similarly to inequalities, the negation of conditions denies the value returned by the truth of the condition. Numeric value comparisons represented a harder challenge as there are intervals where the condition holds. We use the difference of values to represent the error. In the $<$ and $>$ rules, the δ constant depends on the type of the numerical value, 1.0 for integers and 0.00001 for doubles, and is essential for the extra step required for the condition to hold its truth value. A rectifier linear unit was used to ensure that if the condition holds, it is set to the maximum between the negative number and 0, otherwise, if the value is greater than 0, the positive fitness value is normalized.

The fitness function is the result of applying each f_i for each non-liquid refinement to a set of randomly generated (using the liquid synthesis algorithm in Sect. 3.1) input values. The fitness of an individual is the combination of all f_i for all random input values.

4 The RTGP Algorithm

The proposed RTGP algorithm follows the classical STGP [24] in its structure but differs in the details. Just like in all GP approaches, multiple variants can be obtained by changing or swapping some of the components presented here.

4.1 Representation

RTGP can have either a bitstream representation (e.g., [31]) or a direct representation (e.g., [24]). For the sake of simplicity, let us consider the direct representation in the remainder of the paper.

4.2 Initialization Procedure

To generate random individuals, the algorithm mentioned in Sect. 3.1 is used with the context and type of the \blacksquare as arguments. This is repeated until the population has the desired size. Koza proposed the combination of full and grow as ramped-half-and-half [14], which is used in classical STGP. In RTGP, the full method is not always possible, since no valid expression with that given depth

may exist in the language. If, for instance, we want an expression of type X and the only function that returns X is the constructor without any parameters. In this case, it is impossible to have any expression of type X with d greater than 1. Unlike in the STGP full method, a tree is used in the initial population, even if it does not have the predetermined depth.

4.3 Evaluation

The goal of the search problem is the minimization of the error between the observed and the expressed specification. Non-liquid refinements are translated to multi-objective criteria (following the approach explained in Sect. 3.2). The input values are randomly generated at each generation to prevent overfitting [9]. A fitness of 0.0 for one clause represents that all sets of inputs have passed that condition successfully. The overall objective of the candidate is to obtain a 0.0 fitness in all clauses.

4.4 Selection and Genetic Operators

Recent work has provided significant insights on parent selection in program synthesis [11]. A variant of lexicase selection, dynamic ϵ -Lexicase [17] selection, has been used to allow near-elite individuals to be chosen in continuous search spaces.

The mutation operator chooses a random node from the candidate tree. A replacement is randomly generated by providing the node type to the expression synthesis algorithm along with the current node depth, fulfilling the maximum tree depth requirement. The valid subtrees of the replaced node are provided as genetic material to the synthesizer, allowing partial mutations on the candidate.

The crossover operator selects two random parents using the dynamic ϵ -lexicase selection algorithm. A random node is chosen from the first parent, and nodes with the same type from the second parent are selected for transplantation into the first parent. If no compatible nodes are found, the expression synthesizer is invoked using the second parent valid subtrees, and the remaining first parent subtrees as genetic material. This is similar to how STGP operates, with the distinction that subtyping in Liquid Types refers to the implication of semantic properties. Thus, unsafe recombinations and mutations will never occur.

4.5 Stopping Criteria

The algorithm iterates over generations of the population until one or multiple of the following criteria are met: a) there is an individual of fitness 0.0; b) a predefined number of generations have been iterated; c) a predefined time duration has passed.

5 Examples of RTGP

This section introduces three examples from the literature implemented in $\mathcal{A}EON$.

5.1 Santa Fe Ant Trail

The Santa Fe Ant Trail problem is frequently used as a benchmark for GP. In [26], the authors propose a grammar-based approach to solve this problem. In RTGP, if-then-else conditions and auxiliary functions (via lambda abstraction) are embedded in the language, making this a very readable program.

```

type Map;
food_present : (m:Map) → Int = native;
food_ahead : (m:Map) → Boolean = native;
left : (m:Map) → Map = native;
right : (m:Map) → Map = native;
move : (m:Map) → Map = native;
program : (m:Map) → m2:Map where ( food_present(m2) == 0 ) { ■ }

```

Listing 1.3. Santa Fe Ant Trail

5.2 Super Mario Bros Level Design

The second example defines the search for an interesting design for a Super Mario Bros level that maximizes the engagement, minimizes frustration and maximizes challenge. These functions are defined according to a model that can easily be implemented in $\mathcal{A}EON$ (Listing 1.4). We present this as a more usable alternative to the one that uses GGGP [32].

```

type X as {x:Integer | 5 <= x && x <= 95 }
type Y as {x:Integer | 3 <= x && x <= 5 }
type Wg as {x:Integer | 2 <= x && x <= 5 }
type W as {x:Integer | 2 <= x && x <= 7 }
type Wb as {x:Integer | 2 <= x && x <= 6 }
type Wa as Wb
type Wc as W
type Level as Pair<List<Chunk>, {enemies:List<Enemy> | 2 <= enemies.size &&
enemies.size <= 10}>;
type BoxType;
block_coin() → BoxType = native;
rock_coin() → BoxType = native;
block_powerup() → BoxType = native;
rock_empty() → BoxType = native;
type Chunk;
gap(x:X, y:Y, wg:Wg, wb:Wb, wa:Wa) → Level = native;
platform(x:X, y:Y, w:W) → Level = native;
hill(x:X, y:Y, w:W) → Level = native;

```

```

cannon_hill(x:X, y:Y, wg:Wg, wb:Wb, wa:Wa) → Level = native;
tube_hill(x:X, y:Y, wg:Wg, wb:Wb, wa:Wa) → Level = native;
coin(x:X, y:Y, w:Wc) → Level = native;
cannon(x:X, y:Y, wg:Wg, wb:Wb, wa:Wa) → Level = native;
tube(x:X, y:Y, wg:Wg, wb:Wb, wa:Wa) → Level = native;
boxes(t:BoxType, b:{List<Pair<X,Y>| 2 <= b.size && b.size <= 6 }) → Level =
native;
type Enemy;
koopas(x:X) → Enemy = native;
goompas(x:X) → Enemy = native;

generateLevel() → l:Level where ( @maximize(engagement(l)) and
    @minimize(frustration(l)) and @maximize(challenge(l)) { ■ }

```

Listing 1.4. Super Mario Bros Level Design

Compared with the proposed grammar [32], the complexity is similar and productions in either version are directly correspondent. The $\mathcal{A}EON$ version is arguably more expressive because the combinations of repetitions of objects with minimum and maximum number of repetitions can be bounded using types (enemies and boxes).

5.3 Logical Gates

The third example is taken from [27], where the goal is to “given any logical function, find a logically equivalent symbolic expression that uses only the operators in one of the three following complete sets: and, or, not, nand, nor”. The authors propose a Christiansen grammar, which is context-sensitive, to express this problem. Listing 1.5 presents a more simple implementation of the problem using $\mathcal{A}EON$. It can be argued that the implementation using refinements comes more directly from the problem statement than the complex dynamic grammar used in [27]. Furthermore, the implementation of the operations can be done directly in the same language.

```

set(x:Boolean) → y:Boolean = uninterpreted;
andG(x:Boolean, y:Boolean) → z:Boolean where ( set(x) == 1 and set(y) == 1
    and set(z) == 1 ) = { x && y }
or(x:Boolean, y:Boolean) → z:Boolean where ( set(x) == 1 and set(y) == 1 and
    set(z) == 1 ) = { x || y }
not(x:Boolean) → z:Boolean where ( set(x) == 1 and set(z) == 1 ) = { !x }
nand(x:Boolean, y:Boolean) → z:Boolean where ( set(x) == 2 and set(y) == 2
    and set(z) == 2 ) = { !(x && y) }
nor(x:Boolean, y:Boolean) → z:Boolean where ( set(x) == 3 and set(y) == 3 and
    set(z) == 3 ) = { !(x || y) }
target : (x:Boolean, ..., z:Boolean) → e:Boolean where ( e == f(x,...z) ) { ■ }

```

Listing 1.5. Equivalent Logical Gates to a given function f.

6 Discussion

This section compares RTGP with GGGP and presents arguments why RTGP could be used instead of GGGP. Because Dependent Types can encode grammars [5], the performance of both approaches is equivalent.

6.1 A Direct Comparison with GGGP

A survey on GGGP [21] identified the advantages and disadvantages of GGGP. We compare with RTGP on the advantages:

- **Ability to declaratively restrict the search space**—A type system is used instead of a grammar to express the restriction.
- **Problem Structure**—Problem domains that already follow a grammar structure can be easily encoded in RTGP. RTGP can more directly encode several problems than a grammar. Two examples are General-purpose programming and the Logical Gates problem (Sect. 5.3).
- **Homologous Operators**—Both GGGP and RTGP restrict the replacement of one component by another of similar close values.
- **Flexible Extension**—Extensions to GP can be encoded both in grammars and dependent types. Both approaches can be used as engines to test other GP concepts.

And disadvantages:

- **Feasibility Constraints**—Both GGGP and RTGP make the design of new operators a more significant challenge than in STGP, given that operators should follow the constraints imposed by the system. All RTGP operators are shared among any problem and rely solely on two algorithms: the type checker and expression synthesis.
- **Repair Mechanisms**—Implementing repairing in GGGP often depends on the grammar. RTGP relies on an expression synthesis algorithm (Sect. 3.1) that generates individuals in a way that constraints are never violated. The same has been done for the mutation and crossover operators. However, a repair mechanism is straightforward: the type-checker identifies the malign node, and expression synthesis generates a replacement.
- **Limited Flexibility**—GGGP is flexible when the program can be directly encoded in a context-free grammar. Some GGGP approaches use context-sensitive grammars (CSGP) [27], but readability can become a problem (explained in Sect. 6.2).
- **Turing Incompleteness**—GGGP supports grammars with semantics that allow the encoding of Turing-complete and incomplete problems. As such, GGGP does not offer any additional support for computation paradigms such as recursion and iteration, like other GP systems. RTGP supports both recursion and iteration directly, unless otherwise specified.

6.2 Usability

Instead, the main argument for RTGP over GGGP is one that concerns with usability. First, RTGP provides an integrated environment for describing the context, the problem, the search space and the solutions. Taking $\mathcal{A}EON$ as an example of RTGP, the environment in which the final program will execute can be defined, relying on native functions to use software written in other programming languages. The problem is defined using refined types for the goal of the system, and a hole marker (■) is left as a placeholder for the program we are looking for. The search space is defined by the types used in the problem definition. Finally, the solution is a program in the same language as everything else, so it is ready to execute (and be evaluated).

On the other hand, if one were to use GGGP, one would have to create each of these components individually. The lack of a de-facto standard framework for GGGP helps this argument, in which interfacing with the context can be more complicated than implementing GGGP itself. GGGP concerns only with the description of the search space, while RTGP provides an integrated view of using GP.

The strongest point for RTGP is that it does not require the user to define a grammar. Just by placing holes in a program, users can use RTGP without even knowing how to define a grammar. Instead, they need to know how to use a familiar programming language (which to implement GGGP is already required) and to know how to express desired properties in refined types. While refined types have not yet become mainstream, several languages have feature subsets of its features for a long time. Eiffel [23] supported pre- and post-conditions since 1986. Ada is another language that supports design by contract [4], and it is very popular for critical embedded development, being used for large projects in air traffic control [16] with more than 1 million lines of code. Advanced type systems have become more popular to prevent bugs from existing in codebases. Mozilla created Rust to avoid concurrency issues in the Firefox browser [20], and Microsoft is using PL and SMT-based techniques to verify low-level critical components of the kernel and drivers [1].

7 Conclusions and Future Work

We have presented Refinement Typed Genetic Programming (RTGP) as an approach to describe search problems in an integrated programming language. We have introduced a language, $\mathcal{A}EON$, capable of expressing the environment, the fitness function, the search space, and the solution. The language features an advanced type system with liquid and non-liquid types. We have provided a methodology to generate the fitness function from non-liquid refined types, and we have introduced an algorithm that generates expressions from any inhabitable type in this language.

In Sect. 6 we have compared RTGP against GGGP, concluding that they are equivalent in expressiveness. However, we argue that RTGP provides better usability for end-users than GGGP, in which all aspects of the evolution have

to be implemented. Furthermore, expressing restrictions in types allows more modular programs and better readability inside an integrated experience for defining and using RTGP.

There are still some aspects to explore with regards to RTGP: identifying the most efficient representation; improving the liquid type synthesis; finding the best representation for non-functional properties of programs; how to integrate this synthesis in an integrated editor; and to perform an exhaustive benchmark performance analysis.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24756-2_1
2. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_6
3. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. *J. Funct. Program.* **23**(05), 552–593 (2013)
4. Brink, K., van Katwijk, J., Toetenel, W.: Ada 95 as implementation vehicle for formal specifications. In: Proceedings of 3rd International Workshop on Real-Time Computing Systems and Applications, pp. 98–105. IEEE (1996)
5. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 58–79. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13321-3_6
6. Christiansen, H.: A survey of adaptable grammars. *ACM SIGPLAN Notices* **25**(11), 35–44 (1990)
7. Elmas, T., Tasiran, S., Qadeer, S.: Vyrd: verifying concurrent programs by runtime refinement-violation detection. *ACM SIGPLAN Notices* **40**(6), 27–37 (2005)
8. Gissurarson, M.P.: Suggesting Valid Hole Fits for Typed-Holes in Haskell. Master’s thesis (2018)
9. Gonçalves, I., Silva, S., Melo, J.B., Carreiras, J.M.B.: Random Sampling Technique for Overfitting Control in Genetic Programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) EuroGP 2012. LNCS, vol. 7244, pp. 218–229. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29139-5_19
10. Haynes, T.D., Schoenfeld, D.A., Wainwright, R.L.: Type inheritance in strongly typed genetic programming. *Adv. Genetic Program.* **2**(2), 359–376 (1996)
11. Helmuth, T., McPhee, N., Spector, L.: Lexicase selection for program synthesis: a diversity analysis, pp. 151–167 (2016)
12. Korel, B.: Automated software test data generation. *IEEE Trans. Software Eng.* **16**(8), 870–879 (1990)
13. Kosakovsky Pond, S.L., Posada, D., Gravenor, M.B., Woelk, C.H., Frost, S.D.: Gard: a genetic algorithm for recombination detection. *Bioinformatics* **22**(24), 3096–3098 (2006)
14. Koza, J.R.: Genetic Programming: On The Programming of Computers by Means Of Natural Selection, vol. 1. MIT press (1992)

15. Krawiec, K.: Program synthesis. Behavioral Program Synthesis with Genetic Programming. SCI, vol. 618, pp. 1–19. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-27565-9_1
16. Kruchten, P., Thompson, C.J.: An object-oriented, distributed architecture for large-scale ADA systems. In: Proceedings of the Conference on TRI-ADA 1994, pp. 262–271 (1994)
17. La Cava, W., Helmuth, T., Spector, L., Moore, J.H.: A probabilistic and multi-objective analysis of lexicase selection and e-lexicase selection. *Evol. Comput.* **27**(3), 377–402 (2019). <https://doi.org/10.1162/evco.a.00224>
18. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a Java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA), pp. 404–420 (1998)
19. Loveard, T., Ciesielski, V.: Representing classification problems in genetic programming. In: *Evolutionary Computation*, vol. 2, pp. 1070–1077. IEEE (2001)
20. Matsakis, N.D., Klock, F.S.: The rust language. *ACM SIGAda Ada Letters* **34**(3), 103–104 (2014)
21. McKay, R.I., Hoai, N.X., et al.: Grammar-based genetic programming: a survey. *Genetic Program. Evol. Mach.* **11**(3), 365–396 (2010)
22. McPhee, N.F., Hopper, N.J., Reiersen, M.L.: Impact of types on essentially typeless problems in GP. In: *Genetic Programming*, pp. 232–240 (1998)
23. Meyer, B.: Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices* **22**(2), 85–94 (1987)
24. Montana, D.J.: Strongly typed genetic programming. *Evol. Comput.* **3**(2), 199–230 (1995)
25. Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. *Proc. ACM Program. Lang.* **3**(POPL), 1–32 (2019)
26. O’Neill, M., Ryan, C.: Grammar based function definition in grammatical evolution. In: *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, pp. 485–490 (2000)
27. Ortega, A., De La Cruz, M., Alfonseca, M.: Christiansen grammar evolution: grammatical evolution with semantics. *IEEE Trans. Evol. Comput.* **11**(1), 77–90 (2007)
28. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk> (2008), (With contributions by J. R. Koza)
29. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 522–538. ACM (2016)
30. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 159–169 (2008)
31. Ryan, C., Nicolau, M., O’Neill, M.: Genetic Algorithms Using Grammatical Evolution. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A. (eds.) *EuroGP 2002*. LNCS, vol. 2278, pp. 278–287. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45984-7_27
32. Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J., O’neill, M.: Evolving levels for super mario bros using grammatical evolution. In: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. pp. 304–311. IEEE (2012)
33. Shutt, J.N.: *Recursive adaptable grammars* (1999)
34. Solar-Lezama, A.: *Program synthesis by sketching*. University of California, Berkeley (2008)

35. Spector, L., Barnum, H., Bernstein, H.J., Swamy, N.: Finding a better-than-classical quantum and/or algorithm using genetic programming. In: *Evolutionary Computation*. vol. 3, pp. 2239–2246. IEEE (1999)
36. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. In: *ACM SIGPLAN Notices*. vol. 49, pp. 269–282. ACM (2014)
37. Yu, T.: Polymorphism and Genetic Programming. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tettamanzi, A.G.B., Langdon, W.B. (eds.) *EuroGP 2001*. LNCS, vol. 2038, pp. 218–233. Springer, Heidelberg (2001). <https://doi.org/10.1007/3-540-45355-5.17>