1

Program synthesis

In this introductory chapter, we characterize and formalize the key concepts of this book, in particular computer programs. We also define the task of program synthesis and determine the main factors that make it challenging. Finally, we delineate several paradigms of program synthesis, among them genetic programming.

1.1 The nature of computer programs

Computer programs are unique among other mathematical formalisms in embodying *algorithms*, i.e. formal recipes for solving entire *classes* of problems. For instance, the greatest common denominator of *any* pair of integers can be calculated using the same short program. This makes programs fundamentally different from entities that are 'attached' to a specific problem *instance*, e.g., a specific route is a solution to a particular traveling salesperson problem.

Programs exhibit this characteristic because they are able to interact with *data*, or, in other words, respond to *input* with some *output*. This is actually more a necessity than an ability: programs *need* data to act upon. A program that expects an input cannot be launched without it. A deterministic program that does not take any input always produces the same output, which, apart from exotic usage scenarios[1], renders it useless.

A nontrivial program exhibits thus a spectrum of possible *behaviors* that depend on the input to which it is applied. Informally, it *does* something – a phrase that is hardly applicable to salesperson's routes. No wonder we tend to attribute programs with agency, saying that a program 'accepts',

[1] For instance, rather than storing a large raster image of a complex fractal, it may be more memory-efficient to store the program that generates that fractal – a compelling example of Kolmogorov complexity.

K. Krawiec, *Behavioral Program Synthesis with Genetic Programming*,
Studies in Computational Intelligence 618,
DOI: 10.1007/978-3-319-27565-9_1

‘chooses’, ‘waits’, ‘assumes’, ‘guarantees’, etc. Such anthropomorphisms feel natural and will by this token occur in this book, even though this habit has been sometimes criticized [27].

The expressive power of a program is conditional upon the programming language in which it is written. Any Turing-complete programming language is sufficient to express all computable functions, a class capacious enough to embrace most known problems of practical and theoretical interest. Even rudimentary programming languages are usually Turing-complete, including esoteric languages that, for instance, comprise just one instruction [39]. Programs written in such languages can implement most conceivable processes, from elementary arithmetic to simulating selected aspects of human intelligence. In particular, nothing precludes one from writing a program that manipulates other programs – interpreters, compilers, and virtual machines are natural examples of this capability.

program

instruction

In this book, by a computer program (*program* for short) we mean a finite discrete structure composed of elementary *instructions* (or *statements*) and capable of performing computation. The representation of programs that is most natural for humans is source code, i.e. text. For program synthesis, the textual form is redundant and cumbersome to handle, so virtually all approaches work with programs represented as *abstract syntax trees* (AST), abridged structures that contain only the effective elements of programs and omit, among others, the delimiters that separate syntactic structures in source code (like semicolons, parentheses, etc.).

abstract syntax tree

The rules of forming *syntactically valid* (i.e. executable) programs from instructions in a given *programming language* are usually expressed as formal grammars. A grammar distinguishes the programs that belong to a given programming language from those that do not. In this book we consider only syntactically valid programs, and it will be sufficient for us to identify a programming language with a (possibly infinite) set $\mathcal{P}$ of programs and abstract from the particular formalism that determines their validity.

programming language

We write $p(in) = out$ to express that a program $p \in \mathcal{P}$ applied to an input data (*input* for short) in produces an output data out (*output*) as the result of execution. Inputs and outputs may be any formal objects representing certain *types*, either simple (usually scalars, e.g., bits, Booleans, numbers) or compound (usually data structures, e.g., lists, matrices, images). If program input is a tuple, its elements will be referred to as *input variables* and denoted by x_is, i.e. $in = (x_1, \ldots, x_k)$.

domain

The types associated with inputs and outputs determine the *domain* $(\mathcal{I}, \mathcal{O})$ of a program, where $\mathcal{I}$ and $\mathcal{O}$ respectively denote the sets of valid input and output values. The elements of $\mathcal{I}$ form *admissible inputsadmissible input*. For instance, the Boolean domain used in many examples throughout this book includes all programs with signatures of the form $\mathbb{B}^n \to \mathbb{B}$, i.e. $\mathcal{I} = \mathbb{B}^n$

and $\mathcal{O} = \mathbb{B}$, where $\mathbb{B} = \{\text{true}, \text{false}\}$. An input that does not belong to $\mathbb{B}^n$, e.g., a real number, is not admissible for programs in this domain.

As shown by Alan Turing [182], there is no way to determine in general whether a program terminates: the halting property is undecidable. For a non-halting program, it becomes impossible to verify if it returns the desired output for a given input, which is the key property in generative program synthesis (Sect. 1.3). To mitigate this problem, in this book we limit our interest to programs that halt. We also consider only deterministic programs.

1.2 Program synthesis

Writing computer programs is an activity that we habitually attribute to humans. In spite of this, the attempts to automate the process of generating computer programs, viz. *synthesize* them, date back to the early years of computer science and artificial intelligence (see, e.g., [188] and Sect. 1.5).

We define the task of *program synthesis* (*task* for short) as an ordered pair $(\mathcal{P}, Correct)$, where $\mathcal{P}$ is a programming language and $Correct : \mathcal{P} \to \mathbb{B}$ is a *correctness predicate*. Solving a task $(\mathcal{P}, Correct)$ consists in finding a program $p \in \mathcal{P}$ that fulfills $Correct$, i.e.:

program synthesis task

correctness predicate

$$p \in \mathcal{P} : Correct(p), \tag{1.1}$$

(cf. [120]). A program p such that $Correct(p)$ is *correct* and forms a *solution* to a program synthesis task.

Because we adopt a metaheuristic approach to program synthesis (Sect. 1.5.3), it is important to explain how the notions introduced above relate to it. While program synthesis corresponds to *problem* (like the traveling salesperson problem mentioned earlier), a program synthesis task with a specific $\mathcal{P}$ and $Correct(p)$ corresponds to *problem instance* in metaheuristic terminology. The working programs considered by a running synthesis method are potential solutions and by this token are referred to as *candidate solutions*, *candidate programs*, or *search points*. A solution to a synthesis task corresponds to *optimal solution*, which we occasionally refer to as *optimal program*.

candidate solution

optimal solution

The correctness predicate $Correct$ is responsible for telling apart the correct and incorrect programs in $\mathcal{P}$. As we detail later, there are several ways in which program correctness can be verified, i.e. *classes* of correctness predicates. For instance, the class exercised in this book involves confronting a program with tests. A given class of correctness predicate is instantiated by a *task specification* S, e.g., a specific set of tests. In most cases, the reference of $Correct$ to specification will be clear from the context and thus assumed implicit unless otherwise stated. Because we consider only halting programs, the correctness considered here is formally the *total correctness*.

task specification

In algorithmic realization, the mathematical ‘find such that’ statement in (1.1) boils down to ‘generate’ or ‘synthesize’. We find the latter term more adequate, as it emphasizes the fact that programs are assembled from smaller entities (instructions) and that programming is by nature combinatorial. This means that program synthesis lacks the concept of a *variable* and sets it apart from conventional optimization, where candidate solutions are usually fixed-length tuples of such variables. These arguments and past literature [44] incline us to lean toward the term *synthesis*.

discrete search problem

Posed in this way, program synthesis is a *discrete search problem* in the artificial intelligence (AI) sense [156], with *search states* are programs in $\mathcal{P}$. $Correct$ partitions the search space into the *goal states* and the non-goal states, i.e. the programs sought and the remaining ones. In the most conservative formulation, this is the only source of information available to a method that performs program synthesis.

There is however an important feature that makes program synthesis a very special search problem. In conventional search problems, a goal test verifies an inherent property of a search state. For instance, to verify if a board state in the peg solitaire puzzle is a goal state, one checks if the number of pegs remaining on the board is one. In contrast, unless one reaches for *formal verification* methods (which are beyond the scope of this book), the correctness of a program cannot be determined by inspecting its structure, i.e. its source code. A program is correct if it *behaves* in the right way, i.e. if the $\mathcal{I} \to \mathcal{O}$ mapping it meets the requirements defined by $Correct$. Correctness of a program is intermediated by its interpretation (semantics), which is an *extrinsic* property, i.e. it is not explicitly present in the symbols that represent the instructions nor in their combination within a program. This behavioral aspect of program synthesis makes it nontrivial and will reverberate many times in this book.

solvable synthesis task

We consider only program synthesis tasks that are *solvable*. The necessary condition for a task to be solvable is that the programming language is expressive enough, i.e. a finite program that meets $Correct$ can be formulated in that language, i.e. $\exists p \in \mathcal{P} : Correct(p)$. Expressibility of a programming language is however not sufficient to guarantee solving a given synthesis task *with a given method*. A synthesis algorithm can be inherently incapable of visiting some regions of search space due to, e.g., certain *search biases*.

1.3 Specifying program correctness

Program synthesis can be alternatively seen as *translation* of a specification S into a program $p \in \mathcal{P}$ such that $Correct_S(p)$. The key difference between these two entities is that specification is passive, i.e. can only be queried to determine program’s correctness, while the resulting program is active in being executable.

A specification S defines the desired *effect* of computation, and as such can be conveniently expressed using *preconditions*, i.e. conditions that constrain the set of program inputs, and *postconditions*, i.e. conditions that program output has to meet given the input data. Formally, for a program $p : \mathcal{I} \to \mathcal{O}$ and a specification $S = (\mathit{precond}, \mathit{postcond})$:

precondition

postcondition

$$p(\mathit{in}) \equiv \mathit{out} : \mathit{postcond}(\mathit{in}, \mathit{out}), \text{where } \mathit{precond}(\mathit{in}) \tag{1.2}$$

where $\mathit{precond} : \mathcal{I} \to \mathbb{B}$, and $\mathit{postcond} : \mathcal{O} \to \mathbb{B}$. For instance, the specification of a program that calculates the integer approximation of the square root of a nonnegative number n can be phrased using pre- and postconditions as follows:

- $\mathit{precond}(n) = \mathit{integer}(n) \wedge n \geq 0$,
- $\mathit{postcond}(n, m) = \mathit{integer}(m) \wedge n^2 \leq m \leq (n+1)^2$.

Specifying program correctness by pre- and postconditions is common not only in theory [120] but also in practice, as epitomized by the growing popularity of the *design-by-contract* paradigm in software engineering [127, 133], where it is ofter realized using 'requires' and 'ensures' clauses.

design-by-contract

There are two fundamentally different ways in which pre- and postconditions can be verified for a given program. The *formal methods* achieve that without running the program, most commonly by constructing a formal proof of program correctness. The theorem to be proven is in general of the form:

$$\forall \mathit{in} : \mathit{precond}(\mathit{in}) \implies \exists \mathit{out} : \mathit{postcond}(\mathit{in}, \mathit{out}) \tag{1.3}$$

If a *constructive* proof of such theorem can be conducted, it will also determine what is the *out* value that satisfies the postcondition. A side effect of conducting that proof is thus a synthesized program. This approach to program synthesis task is rooted in Hoare logic and formal verification [52] (cf. [120]).

Alternatively, the task can be specified by examples. In that case, the task specification S takes the form of a finite list[2] T of *tests*. Each test is an ordered pair $(\mathit{in}, \mathit{out})$, $\mathit{in} \in \mathcal{I}$, $\mathit{out} \in \mathcal{O}$, where *in* is program input, and *out* is the corresponding *desired output*.

test

desired output

We assume that T is *non-redundant*, i.e. $\nexists (\mathit{in}_1, \mathit{out}_2), (\mathit{in}_2, \mathit{out}_2) \in T : \mathit{in}_1 = \mathit{in}_2 \wedge \mathit{out}_1 = \mathit{out}_2$, and *coherent*, i.e. $\nexists (\mathit{in}, \mathit{out}_1), (\mathit{in}, \mathit{out}_2) : \mathit{out}_1 \neq \mathit{out}_2$. In genetic programming, tests are often referred to as *fitness cases*. A program synthesis task posed in this way can be considered as a machine learning (ML) task defined within the paradigm of *learning from examples*, with T

learning from examples

[2] In GP and machine learning literature, T is typically defined as a *set*. However, maintaining a fixed ordering of tests in T becomes important at certain point of our discourse, so we define T as a list.

playing the role of a *training set* and each test corresponding to an example. As in ML, T is not necessarily assumed to enumerate all possible program inputs; in general, it may be considered a sample drawn from a (potentially infinite) *universe* of tests $\mathcal{T}$.

Given a set of tests $T \subseteq \mathcal{T}$, we can define the correctness predicate as

$$Correct_T(p) \iff \forall (in, out) \in T : p(in) = out, \tag{1.4}$$

target

where $p(in)$ denotes an application of program p to an input data in. The vector of desired outputs out is alternatively referred to as *target*.

Specifying correctness with examples is usually *partial*, because the desired behavior is unspecified for any $in : \nexists (in, out) \in T$. Formal specification is, by contrast, usually *complete* and thus more general. Yet, formal correctness predicates can be difficult to design without a strong mathematical background, and can sometimes be more difficult than writing the program in question. On the other hand, though reasoning in terms of examples is natural for humans, a large number of examples may be required to specify the desired behavior. In the search for alternatives, the notion of specification is being recently extended to embrace other ways of expressing the desired outcome of a synthesis process. In this context, program synthesis can be rephrased more generally as the task of discovering an executable program from *user intent* [44]. Recent interesting developments in this area include expressing intent interactively [45] and writing incomplete programs to be complemented by a synthesis system [167].

specification-driven program synthesis

We propose to group program synthesis paradigms with respect to the workflow they implement. In the top-down *specificiation-driven* approach, it is the specification that drives the synthesis process. A synthesis algorithm starts with the given specification S, analyzes it, and derives (usually deduces) a program from it. The derived program conforms by construction to the $Correct$ predicate, so it does not have to be verified for correctness. Such a workflow is characteristic to, among others, systematic deductive approaches to program synthesis [120] (see Sect. 1.5).

generative program synthesis

In the bottom-up, *generative*, or *generate-and-test* approach, the synthesis process uses a generator of programs that works in a more 'undirected' way. The generated candidate programs are verified using the $Correct$ predicate (which in such case can be considered as a form of *oracle*). The feedback from the verification is subsequently used to produce the next, hopefully better, candidate solution(s). Such generate-and-test workflow is of, among others, genetic programming [79, 148], where an evolutionary algorithm serves as a generator of programs, and program correctness is verified by an evaluation function (Sect. 1.5.3).

An implication of adopting the top-down mode is that a program synthesis method has to 'understand' the specification in order to translate it into a program. In contrast, such a capability is not essential for bottom-up, generative approaches. The latter are thus more domain-independent, and can be conveniently used with complex domain-specific languages, where instructions may be intricate and have complex effect; the languages designed for image analysis may serve as examples here [11, 90]. In a sense, generative approaches assume that the synthesis task in question is too complex to be solved analytically and has to be heuristically 'datamined' to gain some understanding of it, and so facilitate finding a solution. This perspective is congruent with our vision of behavioral program synthesis (Chap. 3), and is one of the reasons why this perspective is built upon the generative stochastic metaheuristic of genetic programming.

1.4 Challenges in program synthesis

There are several reasons why program synthesis is challenging and robust and scalable program synthesizers are yet to be seen. The most obvious one is the size of the search space. The number of combinations of instructions grows exponentially with program length, even if only some of them are syntactically correct in a given programming language. This affects not only the bottom-up methods that need to search that space directly, but also indirectly the top-down approaches, because the size of program space is reflected in the number of paths in the proof space that need to be considered.

As an example, consider the task of synthesizing an m-ary Boolean function $\mathbb{B}^m \rightarrow \mathbb{B}$ represented as an expression tree, in which the programming language comprises k binary (i.e. two-argument) instructions. There are $\binom{n/2}{k}\binom{n/2}{m}cat(n)$ programs represented as trees composed of n instructions, where $cat(n)$ is the nth Catalan number: $cat(n) = \binom{2n}{n}/(n+1)$ [180]. For simplicity, let us assume that the task can be solved using a program that fetches each of m input variables exactly once, i.e. such that forms a binary tree with m leaves and $m - 1$ internal nodes. Even for the moderately difficult 11-bit multiplexer ($m = 11$) [122] and $k = 4$ binary instructions, the above formula results in the staggering 2.93×10^{11} programs – and this is a *conservative estimate* of the size of the search space that needs to be explored.

The second challenge in program synthesis is that programming languages are rich enough to express the same functionality in many ways. Formally, the mapping from the space of programs $\mathcal{P}$ to the space of their behaviors (interpreted for instance as the outputs produced for all tests, like in semantic GP, Chap. 5) is many-to-one. This non-injective characteristic manifests also in the existence of multiple correct programs for a given task, or, put in terms of search problems, in the existence of multiple goal states. When a search problem of program synthesis is recast as an optimization multimodal evaluation function

problem (Sect. 1.5.3), this causes an evaluation function to be *multimodal*. However, this is contingent also on the structure of the search space induced by search operators, as accurately commented for evolutionary algorithms (EA) by Lee Altenberg:

> The multiple-attractor problem is usually described as "multimodality" of the fitness function, but it must be understood that the fitness function by itself does not determine whether the EA has multiple domains of attraction – it is only the relationship of the fitness function to the variation-producing operators that produces multiple-attractors. [2, p. 4]

generalization

On one hand, multimodality increases the statistical odds of finding a solution; on the other, it makes it more difficult to prioritize the search in the presence of multiple potentially useful search directions. Also, multimodality may be a sign of a program synthesis task being *underconstrained*, which is particularly likely when a correctness predicate involves few tests. In such cases, the synthesized program is expected to *generalize* well beyond the training set of tests. This presents a challenge on its own: how to ensure, for instance, that a program meant to calculate the median of a list of numbers, synthesized from a handful of tests, calculates the correct value for any input?

semantics

epistasis

Expressibility of computer programs gives rise to yet another problem. Instructions, the underlying components of programs, are abstract symbols that do not mean anything on their own. Their meaning resides in *semantics*, materialized in the 'substrate' that provides for program execution (be it an interpreter, compiler, or hardware). The semantics of individual instructions is usually simple, and in generic programming languages may comprise little more than elementary logic and arithmetic. Yet, because instructions can be applied in different contexts (e.g., to various arguments, variables, subprograms, etc.) and in various orders, their overall effect is hard to model. As a consequence, the impact of a given instruction on the final computation outcome is highly contextual – the interpretation of a given piece of code in a program depends on its surroundings. This is particularly evident in imperative programming languages (and virtually absent in the functional ones). Put in terms of evolutionary computation, the underlying vehicle of GP, if instructions in a program are likened to genes in a chromosome, then there is strong *epistasis* between them (cf. Sect. 1.5.3).

emergence

The complexity and essential character of interactions between instructions is such a prominent feature of programs that it inclined John H. Holland to use them to illustrate emergence in his seminal work on this topic:

> Interactions play a central role in the study of emergence. A detailed knowledge of the repertoire of an individual ant does not prepare us for the remarkable flexibility of the ant colony. The capabilities of a computer program are hardly revealed by a detailed analysis of the small set of instructions used to compose the program. [56, pp. 38-39]

Indeed, as we argued elsewhere [84, 5], a behavior of a program can be considered its emergent property. Only a part of behavior (pertaining to the program's final outcome) matters for solving a synthesis task; what a program does on its route to producing an outcome is – in a sense – irrelevant. Arbitrarily complex behaviors of programs emerge from a handful of relatively simple instructions. In this light, it is not an overstatement to equate a program, or even more a yet-imperfect program 'in the making', with a complex system [131].

It may be useful at this point to confront the complexity of semantic effects of program execution with a conventional AI-type search problem, the peg puzzle mentioned in Sect. 1.2. The effects of peg moves (modifications of the current solution) directly follow from the board structure and state. They do not refer to any external 'body of knowledge', like semantics of logic or arithmetic instructions in programming. Similar moves (e.g., moves applied to the same peg) have usually similar effects. The evaluation function (the number of pegs left on the board) changes more or less gradually with consecutive moves. Compared to computer programs, this is a really straightforward environment.

The above-mentioned property of similar moves having similar effects is closely related to the notion of *locality* in evolutionary computation (EC) [154]. A problem is said to exhibit high locality if applying a move to a candidate solution leads to solution with similar evaluation. High locality facilitates designing search operators and is usually considered as a sign of a problem's simplicity. Consider a conventional optimization task, the combinatorial traveling salesman problem (TSP). A candidate solution in TSP is an ordering of cities to be visited, encoded as a permutation of natural numbers. Similar permutations in TSP tend to represent similar routes. A move that swaps two edges in a route may affect route length but will not 'ruin' it, as all the remaining edges remain intact.

locality

Computer programs are notorious for being anything but local in the above sense [34, 85]. The mapping from program code to its behavior can be particularly complex: a minute modification of the former may cause a dramatic change in the latter. On the other hand, a major change in a program may be behaviorally neutral. In other words, conventional objective function in GP is known to exhibit low *fitness-distance correlation* [181], i.e. it does not correlate well with the measures of syntactic similarity between programs. This applies to generic distance measures like edit distance (see, e.g., [140]) as well as to operator-based distance measures, like crossover-based distance proposed in [46]. Put in yet another way, fitness landscapes in GP tend to be 'rugged'.

fitness-distance correlation

These properties of programs has been aptly commented by Edsger Dijkstra:

> In the discrete world of computing, there is no meaningful metric in which "small" changes and "small" effects go hand in hand, and there never will be. [27]

James Gleick phrased this characteristics in a more general way, but also more evocatively:

> Computer programs are the most intricate, delicately balanced and finely interwoven of all the products of human industry to date. They are machines with far more moving parts than any engine: the parts don't wear out, but they interact and rub up against one another in ways the programmers themselves cannot predict. [40, p. 19]

In summary, program synthesis is a challenging task due to size of a search space, its multimodality, externalized semantics of instructions, and complex interactions between them. It is thus not surprising that it spawned not one but several research paradigms presented in the next section.

1.5 Paradigms of program synthesis

In this section, we characterize the main paradigms of program synthesis: deductive program synthesis (Sect. 1.5.1), inductive programming (Sect. 1.5.2), and genetic programming (Sect. 1.5.3), the approach used in this book. The former two paradigms are largely top-down according to the taxonomy introduced in Sect. 1.3, while the latter one is purely bottom-up and generative. Rather than providing a complete review, our aim in this section is to position genetic programming in the context of other paradigms.

1.5.1 Deductive program synthesis

deductive program synthesis

In *deductive program synthesis*, one assumes that task specification is complete, i.e. determines the desired output of a sought program for all admissible inputs. The cornerstone of this paradigm is the Curry-Howard correspondence [59], which proves a one-to-one relationship between programs in computer science and proofs in logic. By this virtue, deductive program synthesis boils down to theorem proving, and involves transformation rules, unification, and resolution [120].

The key advantage of deductive synthesis is that the resulting programs are *correct by construction* [28]. On the other hand, its usefulness directly depends on effectiveness of theorem provers, which is nowadays still quite limited. Moreover, achieving complete proof automation is challenging; this is one of the reasons why, for instance, the Coq system, which famously helped proving the four-color theorem, is advertised as a 'proof assistant' rather than a 'theorem prover' [32]. As a consequence, deductive synthesis approaches do not scale well and, depending on the genre, are currently capable of synthesizing programs no longer than a few dozen instructions.

The other challenge for deductive program synthesis stems, paradoxically, from its complete nature. Specifying the desired behavior for all possible inputs is natural for more formal program synthesis tasks, like the square root function considered in Sect. 1.3. However, for many tasks the desired behavior may be not explicitly given. Consider for instance a program that implements a game strategy and should respond with an action (output) to a given board state (input). As the ultimate game outcome is delayed and conditional upon the behavior of an opponent, the most desirable move (desired output) may be not known for a give board state.

Last but not least, even if a complete specification of behavior does exist, it may be cumbersome or difficult for a human programmer to formalize it. It may be thus more natural to express the desired outcome of program synthesis by, e.g., providing examples of desired behavior. Such a process is characteristic of inductive programming as described in the next section.

1.5.2 Inductive programming

inductive programming

Contrary to deductive program synthesis, in *inductive programming* task specification is not assumed to be complete: admissible inputs to a program may exist for which the corresponding output is not given. Specification has the form of a list of tests T, which do not have to enumerate all admissible program inputs (1.4). A synthesis method is expected to perform *induction*, i.e. synthesize a program that does not only passes the tests in T, but also behaves 'accordingly' for the inputs not covered by task specification, i.e. for tests in $\mathcal{T} \setminus T$, where $\mathcal{T}$ is the universe of all tests (cf. Sect. 1.3). What 'accordingly' means depends on the given task and domain, and is often not formalized. For instance, given only a handful of examples of people's full names, a synthesized program may be expected to correctly extract the initials for *any* first, middle, and last name [45].

inductive logic programming

Such formulation of program synthesis entails *generalization* and clearly resonates with learning. Indeed, the primary representative of inductive programming is *inductive logic programming* (ILP, [162, 161]), recognized nowadays as a branch of machine learning (see, e.g., [134, Ch. 10]. ILP deals mostly with logic-based programming languages, in particular Prolog. Main research efforts in ILP focus on learning from relational data, knowledge discovery, and data mining.

program synthesis vs. machine learning

Inductive program synthesis bears also a certain similarity to *learning from examples*, the arguably most popular paradigm of machine learning [134]. In a sense, program synthesis subsumes machine learning, as every (realizable) classifier can be (and usually is) implemented as a computer program. In this context, a machine learning induction algorithm can be treated as a special form of program synthesizer. Nevertheless, the roads of program synthesis and the mainstream of ML parted ways in the 1990s. ML focused on specific (and often non-symbolic and thus non-transparent) representations of hypotheses (like decision trees, decision rules, bayesian networks, etc.), and in exchange for that enjoyed the availability of efficient (though usually heuristic) synthesis algorithms for inducing them. Program synthesis, on the other hand, could not sacrifice its generality (and transparency of the programs) without losing its primary mission. With the advent of strongly non-symbolic paradigms in ML (e.g., support vector machines and more recently deep neural networks), this chasm only got deeper, and today few consider program synthesis as a form of ML.

1.5.3 Genetic programming

genetic programming

Genetic programming (GP) is a stochastic generate-and-test approach to inductive program synthesis [79, 81]. It rephrases program synthesis as an optimization problem and relies on the metaheuristic of evolutionary algorithms [160, 33, 54], arguably one of the few key bio-inspired metaheuristic approaches [168], to iteratively improve candidate programs. Remarkably, GP has been an important paradigm of EC from the early days of this discipline and many pioneering EC studies were dedicated to evolution of *executable structures*[3]. For instance, as emphasized by Mitchell [131, chap. 9], much of John Holland's early work on rule systems [55] was driven by the urge to evolve executable objects.

fitness

GP shares its architectural underpinnings with other incarnations of the evolutionary metaheuristic, like genetic algorithms (GA) and evolutionary strategies (ES). This iterative search procedure, shown in Fig. 1.1, maintains a working set of candidate solutions P called *population.* The elements of P are programs (candidate programs) and are sometimes referred to as *individuals.* Initially, P is populated with randomly generated candidate programs from the programming language of consideration, i.e. $P \subset \mathcal{P}$. The quality of each program $p \in P$ is then assessed using an *evaluation function* (which we will also occasionally call *fitness* for consistency with past work). If evaluation reveals an optimal program p^*, the search is terminated and p^* is returned as the outcome. Otherwise, a *selection operator*

[3] An executable structure needs to interact with some external 'stimulus' for its characteristics to be revealed. This definition embraces conventional programs (Sect. 1.1), but also for instance analog circuits studied by Koza [81].

is applied to P, producing a subset $P' \subseteq P$ of most promising programs called *parents*. Next, *search operators* are applied to the elements of P', resulting in *offspring* candidate programs, which form the next population P to be processed by the subsequent iteration of the evolutionary loop.[4]

What follows then from this description and from Fig. 1.1 is that an evaluation function plays the decisive role in a GP synthesis process. It is in the center of focus of this book and we will come back to it later in this section.

Apart from evaluation, the course of an evolutionary run is determined by a selection operator and search operators. A typical *selection operator* has the signature $sel : 2^{\mathcal{P}} \to \mathcal{P}$ and, when applied to a working population P, selects a well-performing individual from it. In this book we use only *ordinal* selection operators that interpret evaluation as a value on an ordinal scale (not necessarily a metric scale). Such operators can be alternatively termed *non-parametric* [117, p. 45]. The default selection operator in GP is *tournament selection* (TS). TS samples a low number k (usually $k \in [2, 7]$) of candidate solutions from the population and returns the best of them. It became the common method of choice in GP when it has been recognized that selecting solutions proportionally to fitness (*fitness-proportionate selection*) makes it likely for the best-performing candidate solution to dominate the entire population.

selection operator

ordinal selection operator

tournament selection

fitness-proportionate selection

Search operators are typically unary (*mutation*, $\mathcal{P} \to \mathcal{P}$) or binary (*crossover*, $\mathcal{P} \times \mathcal{P} \to \mathcal{P} \times \mathcal{P}$). The role of the former is to introduce minor changes in candidate solutions; the latter should *recombine* the parent solutions so that the offspring share certain 'traits' with them. For instance in so-called tree-based GP (detailed further in this section), mutation may replace a piece of the parent's AST tree with a randomly generated tree, and crossover swap two randomly selected subtrees in parents' ASTs. In GP, mutation and crossover are often used in parallel, so that some offspring stem from the former while some from the latter. In EC terms, these operators together are supposed to provide for *variation*, which, along with selection, forms the two cornerstones of evolution. All these operators are usually stochastic, i.e. two applications of an operator to the same population P will usually result in a different outcome.[5]

search operator

A GP algorithm thus performs a parallel, population-based search, and is by this virtue expected to be relatively resistant to the risk of gravitating to and getting stuck in local optima. For this reason, it subscribes to the category of *global search*.

The above 'vanilla GP' can be modified and extended in many dimensions, for instance by updating the population individual-by-individual (called

[4] Populations and other collections of candidate solutions are formally multisets, but we refer to them as 'sets' for brevity.

[5] Technically, they are thus random variables or, more precisely, *random functions*.

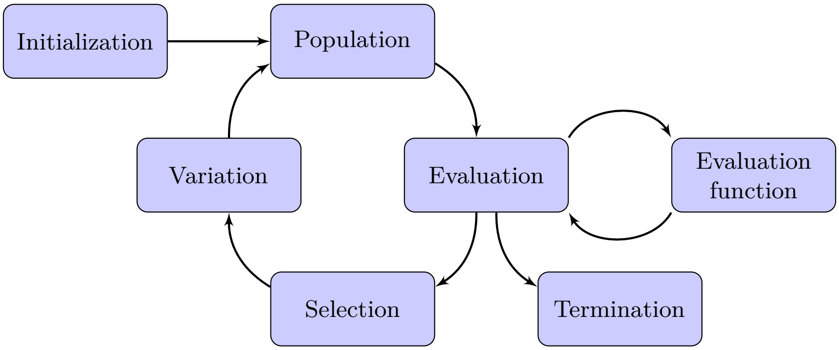

Fig. 1.1: Conventional workflow of genetic programming.

steady-state evolution in contrast to the above *generational evolution*), involving elitism, partitioning the population into *islands*, maintaining an internal *archive* of well-performing candidate solutions, not to mention the panoply of sophisticated selection and search operators. The reader interested in such extensions is referred to textbooks on GP [7, 81, 148] and the online bibliography of GP papers [105].

All those components, however important and beneficial for GP performance, are largely beyond the scope of this book, as our main focus is on the evaluation function, which is arguably the 'root cause' of most decisions made by a search algorithm. In GP, evaluation is based on the performance of a candidate program, i.e. its conformance with the desired behavior as specified by program synthesis task. However, the original formulation of program synthesis as a search problem (1.4) cannot be directly implanted into GP. Evolution, whether natural or simulated, is all about *accretion*, i.e. gradual accumulation of improvements that give individuals a reproductive advantage. It is thus typically assumed that an evolutionary algorithm needs a continuous, or at least multi-valued measure of a solution's quality, i.e. *fitness*, to drive the iterative improvement process. Therefore, virtually all GP genres abandon the qualitative correctness predicate *Correct* in favor of *evaluation function* f with a codomain defined on a scale that is at least ordinal, and usually real-valued, i.e. $f : \mathcal{P} \to \mathbb{R}$. Without loss of generality, we will assume that f is minimized (if not stated otherwise), even though this is somehow inconsistent with the etymology of the term 'fitness'. Nevertheless, to minimize abuse of biological metaphor [168] and for the more fundamental reasons we discuss in Sect. 2.5, we will restrain from using the term 'fitness' unless it is historically justified.

evaluation function

consistent evaluation function

The evaluation functions used in GP are usually consistent with *Correct*, i.e. can indicate an arrival at an optimal solution:

$$f(p) = 0 \iff Correct(p). \tag{1.5}$$

In other words, under a consistent evaluation function, the notion of optimal solution converges with the notion of correct program. Given a consistent evaluation function f, solving a solvable program synthesis task with GP boils down to finding such p^* that

$$p^* = \arg\min_{p \in \mathcal{P}} f(p). \tag{1.6}$$

Contrary to popular belief, we claim that it is not obvious what is the 'right' evaluation function for a given task (or even class of tasks). The formulation of program synthesis (1.1) is agnostic about that. Given a program synthesis task, there will be usually infinitely many evaluation functions that are consistent with its correctness predicate. This observation is important for this book and will ultimately lead us to the concept of *search driver* presented in Chap. 9.

Nevertheless, it is commonly agreed that an evaluation function f should express a program's 'degree of correctness'. GP methods typically calculate such a degree based on program's behavior on tests. Most commonly, f takes the form of

$$f_o(p) = |\{(in, out) \in T : p(in) \neq out\}| \tag{1.7}$$

where T is a nonempty finite list of tests (Sect. 1.3). f_o counts thus the number of tests *failed* (not passed) by p. Alternatively, $f_o(p)$ may count the tests passed by p (a quantity known also as the *number of hits*), in which case it would have to be maximized. In either case, f_o is intended to capture the 'absolute' quality of a program, and by this token is in the following referred to as *objective function*. An objective function is the evaluation function that 'comes with the problem' and is in this sense recognized as the appropriate assessment method of candidate solutions. It is used in GP by default, and to emphasize this fact we will alternatively refer to it as *conventional evaluation function*.

objective function

conventional evaluation function

As signaled in Sect. 1.3, T is often taken from a larger (and sometimes infinite) universe of tests $\mathcal{T}$ and forms in this sense a *training set*. (1.7) becomes in such cases an estimate of the 'true' underlying evaluation, i.e. the fraction of tests passed in entire $\mathcal{T}$.

GP turns the original search problem of program synthesis into an optimization problem. The means by which this is achieved is the *relaxation* of the binary correctness predicate (1.4) into an ordinal evaluation function, in the canonical case f_o. In consequence, GP allows programs to be 'partially correct'. Behind this apparent oxymoron, there is evolutionary rationale related to the aforementioned accretion. Programs that pass only some tests can be iteratively improved and ultimately become correct. Also, exact conformance with the original specification is not critical in some domains. A canonical example is *symbolic regression* , where GP seeks a nonlinear regression model by synthesizing programs that operate in a (typically)

symbolic regression

real-valued domain (i.e. here $(\mathcal{I}, \mathcal{O}) = (\mathbb{R}^n, \mathbb{R})$). The evaluation function commonly used for solving symbolic regression problems with GP is the mean square error (MSE), equivalent up to ordering of candidate programs to the Euclidean distance[6]:

$$f_E(p) = \sum_{(in,out) \in T} (p(in) - out)^2. \tag{1.8}$$

Confronting this formula with f_o (1.7) reveals that f_E 'fuzzifies' the concept of passing a test. This observation will become relevant when defining test-based problems (Sect. 4.1) and program semantics (Chap. 5).

Because we assumed earlier that $\mathcal{P}$ hosts all candidate programs of interest, no additional constraints are necessary to delineate the search space in (1.6), which makes it is an *unconstrained optimization task*. If, for instance, task formulation requires the program being sought to not exceed certain length, we assume that all programs in $\mathcal{P}$ by definition meet such a constraint.

tree-based GP

The way in which a GP algorithm navigates a search space of programs is in part determined by how they are represented. Past GP research delivered several alternative program representations. There is the conventional *tree-based GP*, where programs are represented as expression trees [79], usually equivalent to ASTs. There is the *linear GP*, where programs are sequences of instructions [6, 14]. Another program representations are nested lists of instructions that operate on stacks (*PushGP*, [170]) and graphs of instructions, with edges determining the dataflow between them (*Cartesian GP*, [129]). All these approaches vary only in program representation and conform to the formalisms introduced above.

It should become clear at this point that GP is a methodology that reaches well beyond program synthesis. In contrast to typical formal methods, GP can for instance handle imperfect task formulations (e.g., inconsistent tests) or noisy data. As a consequence, the list of human-competitive achievements of GP is impressive [80, 73]. It is commonly believed that GP's capabilities stem from a combination of two key elements. The first is representing candidate solutions as programs, either conventional or algorithms for classification, regression, clustering, reasoning, problem solving, feature construction, etc. This flexibility enables expressing solutions to virtually any type of problems, whether the task in question is learning, optimization, problem solving, game playing, etc. The second key element is the reliance on the 'mechanics' borrowed from biological evolution, which is unquestionably a very powerful computing paradigm, given that it resulted in life on Earth and development of intelligent beings. This hypothesis, though never scrupulously verified to date, seems to be propelling the interest in and progress of GP.

[6] A GP run that employs tournament selection or other ordinal selection operator will proceed identically for MSE and the Euclidean distance.

1.6 Consequences of automated program synthesis

Once one realizes the capacities of computer programs, it does not take long to notice that the potential consequences of automated program synthesis 'in the large' are profound. Automatically synthesized programs would elevate the robustness of software and implicitly, that of many other technologies. Provably correct programs would make software certifiable, which nowadays can be realized on a very limited scale and only in certain contexts using, e.g., the Coq proof assistant [32]. Automatically generated software would be cheap to produce and malware-free. It could be also paramount with respect to non-functional properties like runtime, memory footprint, or power consumption.

Remarkably, these benefits would stretch beyond the boundaries of programming as currently practiced by humans. Automated program synthesis could help solving tasks that are nowadays either conceptually too complex to tackle, or economically not viable. A particularly useful application is synthesis in programming languages that are difficult and cumbersome for humans but used in practice for all sorts of reasons (legacy, efficient translation into machine language, etc.).

The future of program synthesis can be to some extend foretold by the telltales of current developments. In the following, we touch upon two areas of program synthesis that witnessed remarkable progress in recent years and seem particularly promising: program improvement and end-user programming. It goes without saying that this choice is subjective and other avenues exist, but their full coverage is beyond the scope of this book.

1.6.1 Program improvement

Because synthesizing programs from scratch is challenging (Sect. 1.4), we recently witness growing interest in methods that aim at *improvement* of programs written by humans, more specifically of their non-functional properties like runtime, memory occupancy or power consumption. The key advantage is that a human-written reference program determines the target of the synthesis process. It can be used as a test generator to construct a program synthesis task, or serve as a source of task specification, which can be derived from it using formal methods (e.g., [19]). The former usage is particularly valuable when supply of tests is limited, which is common in some branches of program synthesis [45].

program improvement

non-functional properties

Improvement of non-functional properties has been approached on various abstraction levels. On the level of machine language, it relates to *rewrite systems* studied in compiler design and code optimization. For instance, in [158], Schkufza et al. employed the Markov Chain Monte Carlo technique to improve the runtime of programs written in machine code for a 64-bit

rewrite systems

x86 processor. The reference program is machine code written by a human or compiled from a higher-level language. The Metropolis-Hastings algorithm is used to stochastically generate new candidate programs from it. The authors employ search operators similar to mutations in GP, randomly modifying instructions' opcodes, operands, replacing entire instructions, or swapping them within a program. The optimization is driven by an evaluation function that returns a weighted sum of estimated program runtime and Hamming distance between the desired and actual output for a set of tests. The experiment conducted on the benchmarks taken the famous *Hacker's Delight* volume [189] show an almost systematic reduction of runtime (up to 40 percent), often accompanied with shortening of the resulting code (e.g., from 31 to 14 lines in the case of of Montgomery multiplication procedure). Remarkably, the observed speedups improve over the conventional compilers run with the most intense optimization (e.g., `gcc -O3`).

At a higher abstraction level, Langdon et al. developed a GP framework for manipulating source code written in C++ and applied it to several domains. In [146, 147], they optimized the code of MiniSAT, a popular Boolean satisfiability (SAT) problem solver and obtained accelerations of execution greater than those elaborated by human programmers. In [107], they achieved up to six-time reduction of execution time of a computer vision procedure (stereo disparity estimation algorithm) written for the CUDA architecture running on GPUs. In [108], they reported over 35 percent speedup of registration procedures for magnetic resonance imaging.

At an even higher abstraction level, Kocsis and Swan [77] proposed a more formal method that operates on ASTs and exploits the knowledge of data types to improve programs. By making use of the well-known Curry-Howard isomorphism between proofs and programs [59], they replaced a (traditionally stochastic and non-semantics-preserving) GP mutation operator with deterministic proof-search in the *sequent calculus*. They showed how this operator can be used to automatically replace the singly-linked implementation of a list with the more efficient implementation of a difference list. On the implementation side, they used the reflection mechanism built-in to the Scala programming language to search for amenable data types and accordingly modify the AST trees of the original source code. The semantics-neutral character of this method makes it potentially applicable not only in GP (and in typed GP in particular), but also in the formal and deterministic methods of program synthesis.

1.6.2 Hybrid and interactive program synthesis

In its canonical formulation, program synthesis proceeds in an 'off-line' mode: a user prepares the specification, chooses the programming language (or designs an ad-hoc one), passes them to the synthesis method, and waits for a program to be synthesized. At the current state of advancement of program synthesis, such usage scenario turns out to be far from realistic for programs longer than toy examples. As contemporary techniques do not scale well, preparing a specification and program synthesis may together require more time than writing the program manually.

In response to this, *hybrid* and *interactive* approaches to program synthesis have recently gained more attention. An example of the former can be *sketching* [167], where a user writes a *partial program*, i.e. a program that is missing pieces of code while being otherwise syntactically correct. The method fills in the gaps with pieces of code that complement the partial program so that it becomes correct. By sharing the process of program between a human and a machine, sketching intends to lower the computational complexity of program synthesis.

Interactive approaches to program synthesis assume that a human operator is willing to aid the synthesis process at selected stages. This is particularly useful in *end user programming*, intended to support users with limited programming capabilities. In such application scenarios, one often cannot assume that a user has *any* level of understanding of programming languages. A recent example is here Flash Fill [45], a technology recently developed at Microsoft™ Research and deployed in the 2014 edition of Microsoft Excel™. Flash Fill allows a user to specify a desired transformation of data in a spreadsheet by providing a few examples of what is the desired effect of that transformation. Based on those examples, Flash Fill synthesizes an ad-hoc data transformation program in a domain-specific language, and applies that program to all data entries. By inspecting the outcome and possibly correcting it, the user provides a more detailed feedback for the method, which is used to fine-tune the synthesized program. Internally, Flash Fill relies on a carefully customized domain-specific programming language and uses machine learning techniques to select the hypotheses (candidate programs) that are most likely to meet user expectations.

end user programming

1.7 Summary

In this chapter, we characterized the key properties of programs, presented and formalized the task of program synthesis, and delineated its main paradigms. We also identified the main challenges one faces when attempting to synthesize programs automatically. These challenges limit the capabilities of program synthesis methods. However, we claim that this is in part due to certain design choices that are commonly followed in the generative methods like GP. In this book we focus on the limitations pertaining to the way a search algorithm is informed about the qualities of working solutions. The next chapter is entirely devoted to this aspect.