Behavioral code reuse

To this point, our attempts to widen the evaluation bottleneck focused on defining alternative evaluation functions, which we conceptualize as search drivers in Chap. 9. However, an analysis of an execution record (Chap. 3), whether conducted with information-theoretic measures (Chap. 6) or machine learning algorithms (Chap. 7), also reveals information about the qualities of particular components of candidate solutions, i.e. subprograms. In this chapter, we elaborate on this observation and propose a means to harness its potential. We show how the detailed information available in an execution record together with behavioral evaluation enables (i) identification of useful components of programs, which can be then (ii) archived and (iii) reused by search operators. The following sections detail these stages as realized in [96].

8.1 Identification of useful subprograms

Many machine learning classifiers perform internal feature selection by deciding which attributes to use to construct a classifier. Some of them explicitly reveal that information. This in particular applies to the classifiers that represent their hypotheses (models) symbolically, among others decision trees used in the previous section. For instance, the decision tree in Fig. 7.5 engages only two attributes \( x_1 \) and \( x_4 \) of the four available in the dataset. Similar transparency holds for rule-based classifiers and other more or less ‘white-box’ symbolic representations like Bayesian networks. Nevertheless, the non-symbolic representations do not preclude such possibility: the weights of a neural network can be for instance used to quantify the relative importance of attributes (or even construct a corresponding symbolic representation [29]), and statistical tools exist to assess the importance of attributes in regression models (see e.g., [3]). In summary, almost every type of classifier can be used to estimate the relevance of attributes in a given supervised learning task.
In PANGEA (Chap. 7), behavioral evaluation of a program results in an ML classifier that serves as a basis for calculating behavioral evaluation functions. The classifier is induced from a dataset built upon an execution record. The attributes in that dataset correspond to columns in the original execution record, which in turn point to concrete instructions in the program. This chain of dependencies leads thus to instructions that are judged as most promising by an ML inducer, i.e. producing intermediate results that relate to the desired output. Behavioral evaluation can be thus seen as a means of solving the credit assignment problem [130]: deciding which components of a compound candidate solution are responsible for its overall performance and how to distribute the total reward between them.

On the face of it, considering credit assignment in the context of program synthesis is disputable. Intricate interactions between instructions make it difficult to treat programs as anything but monoliths (c.f. Sect. 1.4). For instance, how much credit should the second line of the program in Fig. 7.1a receive in return for the overall performance of this program (which, by the way, is incorrect as a whole, making this question even harder)?

This question is difficult if not void when asked at the level of source code. However, considerations about credit assignment become appropriate on higher abstraction levels involving structures of code, i.e. blocks, procedures, or in general modules. Because modules typically have well-defined interfaces with the other components (e.g., function signatures) and certain roles in the context of entire program, one might quantify the degree to which a given instance of a module meets the expectations of the context. Put in terms of studies on modularity, such structures are nearly-decomposable but usually not separable (see [163, 190] and Sect. 11.1).

In tree-based GP, the natural interpretation of such a structure is a subtree in program tree, which we refer to in the following as subprogram. If a particular programming language is functional, there is no global state and subprograms perform independent computation. In such a case, the classifier in PANGEA, by referring to a particular column in an execution record, indicates usefulness of the subprogram rooted in corresponding node of program tree. As a side effect of program evaluation, we obtain a subset of useful subprograms. PANGEA is not the only approach in this book where evaluation can adopt this role of ‘subprogram provider’. The trace consistency evaluation function $f_{TC}$ in Chap. 6 has similar capability: formula (6.4) seeks the column in an execution record that minimizes the two-way conditional entropy, and that column pinpoints the subprogram that behaves most consistently with the desired output.

Given a program $p$, the subprograms identified by a classifier $C$ in $p$ will be denoted by $P_C(p)$. We allow $P_C(p) = \emptyset$, i.e. an evaluation process may refuse to indicate any subprograms in $p$ as valuable. In PANGEA, this special case occurs when a classifier does not use any attributes from the dataset.