ADFs: An Evolutionary Approach to Predicate Invention

C. Giraud-Carrier*, C.J. Kennedy

*Department of Computer Science, University of Bristol, Bristol, UK 1Defence Evaluation and Research Agency, EX Building, St Andrew's Road, Malvern, UK

Abstract

This review paper makes explicit, for the first time, the conceptual similarity between Genetic Programming's ADFs and Machine Learning's invented features/predicates, and shows how the evolutionary nature of ADFs allows the complex when and what questions of predicate invention to be answered automatically.

1 Introduction

In machine learning, language biases applied to the hypothesis language restrict the number of expressible hypotheses. Whilst desirable from the point of view of efficiency and generalisation [16], such biases may hinder effectiveness as, in some cases, there will not exist a hypothesis consistent with the training set. For example, it is not possible to express relationships between attributes in the commonly-used attribute-value language. In such cases, the (biased) hypothesis language must be extended with appropriate constructs (e.g., attributes, functions). In fact, even when the hypothesis language need not be extended, it may still be advantageous to do so in order, for instance, to reduce the size of induced hypotheses.

The problem of deciding 1) when new constructs would be useful/necessary and 2) what the new constructs should consist of has become known as constructive induction or predicate invention [14, 29, 20]. The when question of predicate invention is undecidable in the general case [26]. The what question, although rather trivial in principle, begs the more difficult question of utilisation, i.e., which of many potential constructs are most appropriate [20]. Hence, progress has been slow within mainstream machine learning and has largely ignored related work on Automatically Defined Functions (ADFs) in the Genetic Programming (GP) community.

ADFs are an extension to GP involving the evolution of extra functions or modules that support code reuse. In the simplest case, the basic program tree of GP is enhanced to become a structure combining a main program and the subroutines, or ADFs, that may be used by that main program. The ADFs and the main program are evolved simultaneously. In some implementations, λ-abstractions are used to denote "anonymous" ADFs, that are evolved as arguments to higher-order functions in the main body of the program structure. There is a strong link between ADFs (and their extensions) and predicate invention since they address similar problems. Furthermore, the evolutionary nature of ADFs allows the when and what questions of predicate invention to be answered automatically. This review paper makes explicit, for the first time, the natural connection between predicate invention and ADFs.

2 Predicate Invention

We focus here on predicate invention within Inductive Logic Programming (ILP), as this setting subsumes the attribute-value one and is most closely related to the GP setting discussed later. In ILP, predicate invention is the process of introducing new predicates (i.e., functions into \{0,1\}) into the hypothesis language.

There are two motivations for predicate invention, namely, necessity and usefulness. They are formalised in the following definitions from [12]. An invented predicate is necessary if "... without inventing such new term(s), the theory is not learnable according to the criterion of successful learning." An invented predicate is useful if "... its invention does not affect the learnability of the theory." These two types of invented predicates correspond to the two classes of predicate invention approaches identified in [26], namely, demand-driven approaches and reformulation approaches.

Reformulation approaches to predicate invention introduce new predicates to allow reformulation of an existing theory in order to express it in a more compact and concise way. The most commonly used technique is based on inverse resolution and implemented using the intra-construction operator
[17, 19], defined as follows: Given $h_1 \leftarrow \alpha \beta$ and $h_2 \leftarrow \alpha \gamma$, construct/invent $h_3$ such that $h_1 \leftarrow \alpha h_3$, $h_3 \leftarrow \beta$ and $h_3 \leftarrow \gamma$. The new predicate $h_3$ may be either accepted (and named adequately) by an oracle or rejected. One experiment based on Bacon’s *Novum Organum* shows how the learning system CIGOL combines several existing laws about illumination and light sources to construct a new predicate for the concept of radiation [18], whilst experiments with the KPa7KR chess problem boast a 20% compaction using predicate invention [17]. Learning systems that carry out predicate invention using variants of inverse resolution include CIGOL [21], LFP2 [27], ITOU [23], RINCON [30], and Banerji’s system [2]. Other systems, such as CIA [3] and FOCL [24], carry out reformation using schemes, i.e., descriptions of useful literal combinations.

Demand-driven approaches to predicate invention attempt to detect situations where the hypothesis language is insufficient to produce a hypothesis that is consistent with the training examples, i.e., a consistent finite axiomatisation of the examples [12, 26]. For example, Ling reports on an experiment involving learning multiplication (i.e., $\text{mul}(x, y, z)$ is true if $z = xy$) using only the successor function $s$ and the constant 0 [12]. With the given language, $\text{mul}$ is not finitely axiomatisable and the new predicate $\text{foo1}(x, s(z1), s(z)) \leftarrow \text{foo1}(x, z1, z)$, easily recognised as the definition of addition (i.e., $z = x + z1$), is invented. As determining that a hypothesis language is unable to produce a consistent finite axiomatisation of the examples is undecidable (unless the language and algorithm biases allow the full enumeration of the search space) [26], the decision about introducing new predicates must be driven by heuristics. Several learning systems use an over-general clause that can not be specialised with the existing hypothesis language as an indication that predicate invention is desirable. Of course, if over-generality is due to noise, inventing new predicates leads to overfitting. Learning systems that implement a demand-driven approach to predicate invention include INPP [13], DBC [8] and SIERES [28].

An interesting variant of predicate invention is found in the work on repeat learning [7], where predicates learned in a particular context are re-used in further learning tasks. In *intra* repeat learning, the learning task remains unchanged but is performed more than once. Hence, the predicates learned or invented on a given learning attempt are added to the hypothesis language before the next attempt. In *inter* repeat learning, the learning task changes and the predicates learned or invented while learning one task are added to the hypothesis language of another related learning task. These ideas are illustrated in [7] using the well-known chess end-game domain. It is interesting, in the light of this paper, to notice that both intra and inter repeat learning implement a form of code reuse.

3 Automatically Defined Functions

Automatically Defined Functions (ADFs) extend the traditional concept of sub-routines (or modules) to Genetic Programming (GP) [9]. The basic GP program tree is enhanced to become a structure combining a main program (i.e., one value returning branch) and the sub-routines or ADFs (i.e., one or more function defining branches) that are to be used by the main program. The bodies of the ADFs and of the main program are evolved simultaneously using suitably adapted genetic operators.

In practice, the use of ADFs typically leads to smaller, more general solutions to problems of a hierarchical nature, and tends to facilitate the production of solutions to problems challenging for ordinary GP (e.g., Even K-Parity for $k=11$ [10]). In what follows, we give a brief overview of some of the most popular variants of ADFs.

**Evolutionary Module Acquisition** provides a general purpose technique for creating modules in any evolutionary based system [1]. Components of an evolving individual are chosen at random and collated into a module by a compress operation, which is isolated from further manipulation by the genetic operators. An expansion operation can be used to release components from a module in order that they may be manipulated by the genetic operators again. Modules survive based on fitness.

**Automatically Defined Macros** are evolved simultaneously with the main program [25]. A macro is an instruction that is replaced by a sequence of instructions prior to the compilation of the program. Macros are similar to subroutines since they allow programs to be modularised. However, they have the further advantage of allowing new control structures to be implemented through their influence over the evaluation of their arguments.

**Adaptive Representation Learning** is an alternative to ADFs that abstracts useful subtrees from program trees so that they may be used as subroutines in future populations [22]. A useful block of code is identified by looking at an individual’s differential fitness compared to its least fit parent and block activation (i.e., number of times a particular block of code is activated during evaluation). Blocks of code with high activity in individuals with high
positive differential fitness are generalised by replacing a random subset of terminals with variables, and placed as a new subroutine in the subroutine set. All subroutines are assigned a utility value according to the outcome of their use. Subroutines with low utility are deleted from the subroutine set to keep the size of the set below a specified threshold.

“Demand-driven” ADFs extend the standard use of ADFs by providing special operators that allow the individuals in the population to develop their own architecture (i.e., create/duplicate/delete ADFs and ADF arguments) [11]. The special operators are applied to a small percentage of the current population in conjunction with the standard genetic operators. Thus, subroutines and their arguments can be added or deleted on demand to fit the particular problem being solved.

λ-abstractions are used to denote anonymous functions where the name and type are not specified [31, 6]. λ-abstractions are evolved as arguments to higher-order functions in the main body of the program structure. They therefore evolve dynamically and, as with demand-driven ADFs, they are not a predetermined condition of evolution, but are evolved if helpful to the solution. The number and type of arguments of a λ-abstraction are implicitly decided by the type signature of the higher-order function of which they are a functional argument.

The above techniques allow sub-routines to be co-evolved with and used in a main program, thus producing solutions that are more compact and more general. For example, using λ-abstractions, PolyGP evolved a solution for the N-parity problem for any N [31]. In addition, the system invented the XOR, XOR and XAND functions in the evolved solutions. Similarly, STEPS discovered useful properties in carcinogenic chemical compounds [5].

4 Predicate Invention and ADFs - The Missing Link

As seen in section 3, ADFs, and their variants, are used in conjunction with GP to reformulate the main evolving program. The co-evolved subroutines compress the amount of code required in the main program and can also facilitate the discovery of solutions to otherwise challenging problems.

Hence, ADFs clearly fulfill for GP a role similar to that of the reformulation approach to predicate invention in ILP. In fact, because of the iterative nature of the evolutionary process, a natural form of intra repeat learning takes place with ADFs, since the ADFs discovered in one generation become part of the alphabet in the next if they are preserved. Similarly, ADFs can be used for inter repeat learning, where the ADFs evolved for a particular problem are added to the alphabet of new related problems. Interestingly, well-known evolutionary techniques, such as seeding [15] and Species Adaption Genetic Algorithms (SAGA) [4], also implement both intra and inter repeat learning.

The correspondence between ADFs and the demand-driven approach to predicate invention is less trivial. ILP grounds its distinction between usefulness and necessity of predicate invention in logic theory, i.e., finite axiomatisation. GP makes no such distinction explicitly and introduces new functions/sub-routines heuristically based on fitness considerations. Hence, it is difficult to establish whether a sub-routine is created because it is strictly necessary or simply because it is useful. However, one can argue that the distinction between usefulness and necessity is of no real practical interest and of only limited theoretical value since it cannot be decided in general (see section 2). What seems most relevant for practitioners is that their system be capable of inventing predicates/functions whenever this proves valuable (be it useful or necessary) in finding a solution. In this light, it is clear that the ADF approach of GP is the same as the predicate invention approach of ILP. Both must rely on heuristics when it comes to answering the when question.

It is our contention that the GP approach to ADFs (and variants), which is driven solely by fitness considerations, offers a uniform way of treating both types of predicate inventions. Since there is no principled way of inventing predicates, the genetic paradigm, as in many similar circumstances, provides a valuable alternative. The recent approaches based on demand-driven ADFs and λ-abstractions seem most promising, as they leave the answers to the questions of what should be invented, and when it should be invented to be themselves learned thereby subsuming the implementations of both the reformulation and demand-driven approaches to predicate invention in ILP.

5 Conclusion

This paper reviews predicate invention and automatically defined functions and, for the first time, highlights the conceptual similarity between the two concepts. It further shows how the evolutionary nature of ADFs allows the complex when and what questions of predicate invention to be answered automatically. Experiments with benchmark problems are still needed to validate this claim. In particular, will the ADFs produced be predicates in a sense that
ILP researchers will recognise and be interested in? Preliminary results with PolyGP [31] and STEPS [5] are encouraging.

References