Machine learning functional programs

Andrei Diaconu
Somerville College
University of Oxford

Part B Project

Trinity 2020

Abstract

Inductive programming is a field whose main goal is synthesising programs that respect a set of examples, given some form of background knowledge. There are two main areas of research: inductive logic programming (ILP) and inductive functional programming (IFP). In this project, we construct an IFP system, but we draw ideas from the ILP system Metagol [2]. We try to give the problem a formal setting, making analogies to ILP. We also inspect whether the synthesized functions could be reused, something that has not really been researched before. We compare out system to other systems.

Contents

Chapter 1

Introduction

1.1 Inductive programming

Inductive programming (IP) - also known as program synthesis or example based learning - is a subfield that lies at the intersection of several computer science topics (machine learning, artificial intelligence, algorithm design) and is a form of automatic programming. As opposed to deductive programming [10], another automatic programming approach, where one starts with a full specification of the target program, IP tackles the problem starting with an incomplete specification and tries to generalize that into a program [1]. More often than not, that incomplete specification is represented by examples, so we can informally define inductive programming to be the process of creating programs from examples using a limited amount of background information.

AC: Just add a bit of fluff text around this example to introduce it a bit smoother

Example 1 (Possible instance of the IP problem)
Input: The definitions of *map* and *increment* and the examples $f([1,2,3] = [2,3,4])$ and $f([5,6]) = [6,7])$.
Output: The definition $f = map\ increment$.

AC: The appeal of ILP .... ¡- that text does not sound right. IP is not a ML technique, more of a problem. I would rephrase as "One of the key challenges of IP is the need to learn from small numbers of training examples, which mostly rules out statistical machine learning approaches, such as SVMS and neural networks ..."

The appeal of IP, as opposed to other ML techniques, lies in the fact that IP approaches require only a handful of examples and are completely transparent. This however, does not come without pitfalls: if the examples are not representative enough, we might not get the program we expected. The two main directions in

IP have been inductive logic programming (ILP) and inductive functional programming (IFP), but a lot of the core ideas are shared between the two.

As noted in the survey by Gulwani et al [6], one of the main areas of research in IP has been end-user programming. More often than not, an application will be used by a non-programmer, and hence that user will probably not be able to write scripts that make the interaction with that application easier. IP tries to offer a solution to that problem: the user could supply a (small) amount of information, such as a list of examples that describe the task, and an IP system could generate a small script that automates the task at hand. Perhaps one of the most noteworthy applications of this idea is in the *MS Excel* plug-in *Flash Fill* ([5]). Its task is to induce a program that generalizes some spreadsheet related operation, while only being given a few examples (e.g. separate the first name from the second name if the full name is given as input). AC: Give an example of flashfill Another researched area is computer-aided education and ways in which IP can help improve programming skills. As noted by Gulwani [4], the example based background information used in IP shares some similarities to how humans learn from examples. Hence, tasks such as problem generation, solution generation and feedback generation could be automated using IP techniques.

1.2 Motivation

Talk about function/predicate invention AC: Talk about predicate and function invention

Talk about function/predicate reuse AC: Talk about predicate and function reuse

While some ILP systems have explored the idea of reusing functions already synthesized during the program induction process (such as *metagol* and [WHAT OTHER SYSTEMS HAVE?]), AC: Metagol, Hexmil, and to a lesser extend DILP and ILASP. function reuse in the IFP context has not been thoroughly explored [CAN I MAKE SUCH A CLAIM?]. AC: I make the claim in my thesis, which you can cite http://andrewcropper.com/pubs/phd-thesis.pdf (page 19)

AC: Moreover, even though predicate invention and reuse has been claimed as useful, to our knowledge, there has been no work that empirically demonstrates that it is, nor any work discussing when it may be useful. To address this limitation, in this work, we are interesting in the following research questions:

- AC: can function reuse improve learning performance, both in terms of predictive accuracies and learning times
- is modularity (breaking the induced program into lots of small programs) of the induced programs worth then additional computational time, and how does it help with function reuse? AC: not interesting
- can it speed up the computation time by finding smaller programs? AC: stated above
- how important is the allowed grammar of the induced programs, and how does it restrict function reuse? AC: not interesting
- what classes of programs benefit from it? AC: imprecise

AC: The answer to this question is not obvious. Although one can easily construct scenarios where function reuse reduces the size of a program (which should make it easier to learn (CITE some of my papers), it is unclear whether the additional computation time is too much ¡- make this sound better

Another point that prompted the development of our system is that most systems rarely provide a straightforward way to to change the background knowledge that is used when inducing programs. This in turn creates two problems: it makes it harder for the end user to interact with the system and it hinders experimentation. Those two points (function reuse and easily changeable background knowledge) represent the starting point for the system we have developed. AC: not interesting

1.3 Contributions

AC: you main contribution is knowledge, not an implementation, i.e. the point of the work is to explore whether (and when) function reuse helps. You should rephrase this bit.

AC: your second contribution is the implementation, and is simply a necessity to explore your idea. What you can do is add a table which compares your system with Metagol, Magichaskeller, and Lambasquared. You can then clearly show what your system does differently

One of our main contributions is the introduction of a new system, called *FReuse*, which draws ideas from existing systems, but uses a modular approach for inducing programs. This in turn allows us to test the function reuse hypothesis and

explore its implications. We adopt notation used in *metagol*, and draw from ideas present in other system ([3], [7]). The system is used as a testing ground for the function reuse hypothesis, but it also creates an environment in which new ideas can be tested depending on what background information is supplied. Another point we explore is how the supplied knowledge affects the learning process and makes function reuse useful.

1.4 Structure of the report

The rest of the paper is structured as follows:

- **chapter 2**: presents background on IP and a variety of other systems.
- **chapter 3**: presents a formal framework for describing the inductive programming problem.
- **chapter 4**: presents our algorithm, which represents the backbone of *FReuse*, in light of the description presented in chapter 3.
- **chapter 5**: presents implementation details and a detailed example of how *FReuse* induces programs.
- **chapter 6**: explores the role of function reuse and the general capabilities of the system as well as its possible uses.
- **chapter 7**: presents the conclusions, limitations, possible improvements and extensions of the project.

Chapter 2

Background and related work

In the previous chapter, we have informally introduced the concept of IP, presented its relevance and showcased our ideas. In this chapter, we first provide the reader with more background on IP (areas of research, approaches). We then switch to literature review, showcasing different IP systems and their relevance to ours. We finish the chapter by talking about the idea of function reuse.

2.1 Background on IP

Inductive programming has been around for almost half a century, with a lot of systems trying to tackle the problem of finding programs from examples from different angles. It is a subject that is placed at the crossroad between cognitive sciences, artificial intelligence, algorithm design and software development [9]. An interesting fact to note is that IP is a machine learning problem (learning from data) and moreover, in recent years it has gained attention because of the inherent transparency of its approach to learning, as opposed to the black box nature of statistical/neuronal approaches [13].

IP has two main research areas:

- Inductive functional programming (IFP): IFP adresses the synthesis of recursive functional programs generalized from regularities in input-output examples [6], using functional background information. Typically, the target language is a declarative one such as Haskell or Lisp.

- Inductive logical programming (ILP): ILP started as research on induction in a logical context [6]. It's aim is to construct a hypothesis h which explain examples E in terms of some background knowledge B [12].

As highlighted in the review by Kitzelmann [9], there have been two main approaches to inductive programming (for both IFP and ILP):

- **analytical approach**: it's aim is not to exhaustively search for programs, but rather to exploit features in the input-output examples; the first systematic attempt was done by Summers' *THESIS* [14] system in 1977. He observed that using a few basic primitives and a fixed program grammar, a restricted class of recursive LISP programs that satisfy a set of input-output examples can be induced. His main idea was split into two parts: first, transform the examples into non-recursive programs (fragments or traces) that represent an approximation of each input-output example; then, create a recursive program that generalizes over those small programs. Because of the inherent restrictiveness of the primitives, the analytical approach saw little innovation in the following decade, but systems like *IGOR1*, *IGOR2* [8] have built on Summers' work. Another analytical approach uses the idea of *version space algebra*. Those work by analysing input-output traces and creating different hypothesis spaces, which are then combined using algebraic operators to induce programs. The analytical approach is also found in ILP, a well known example being Muggleton's *Progol* [11].

- **generate-and-test approach (GAT)**: with the advent of more powerful computers, enumerating the program space became a possibility; in GAT, examples are not used to actually construct the programs, but rather to test streams of possible programs, selected on some criteria from the program space. Compared to the analytical approach, GAT tends to be the more expressive approach, at the cost of higher computational time. Indeed, the *ADATE* system, a GAT system that uses genetic programming techniques to create programs, is one of the most powerful IP system with regards to expressivity [9]. Another well known GAT system is Katayama's *Magic Haskeller* [7], which uses type directed search and higher-order functions as background knowledge.

Usually, a systems will use a mixture of the two approaches, so as to explore multiple avenues for pruning. Our system is mainly GAT, but we explore and experiment with a some ideas commonly used in analytic approaches. We use a type-inference driven approach to pruning, and explore the possibility of example propagation as a means to disregard unwanted programs.

2.2 Related work

We now present three systems that helped us develop our ideas and hypotheses and contrast them with our work.

2.2.1 Metagol

Metagol [2] is an ILP system that induces Prolog programs. It uses an idea called MIL, or meta-interpretative learning. MIL learns logic programs from examples and background knowledge by instantiating metarules. We present the three forms of background information in more detail:

- compiled background knowledge (CBK): those are small, first order Prolog programs that are deductively proved by the normal interpreter (think of the ancestor relationship).

- interpreted background knowledge (IBK): this is represented by higher-order formulas that are proven with the aid of a meta-interpreter (since Prolog does not allow clauses with higher-order predicates as variables); for example, we could describe *map/3* using the following two clauses:
  $map([],[],F):-$ and $map([A|As],[B|Bs],F):-F(A,B),map(As,Bs)$

- metarules: those are rules that enforce the form of the induced program's clauses; an example would be $P(a,b):-Q(a,c),R(c,b)$, where upper case letters are existentially quantified variables (they will be replaced with CBK or IBK).

The way the hypothesis search works is as follows: try to prove the required atom using CBK; if that fails, fetch a metarule, and try to fill in the existentially quantified variables; continue until a valid hypothesis (one that satisfies the examples) is found. To note here is that *Metagol* propagates examples, i.e. if we have say *map* and some examples, we can infer a set of examples for the third argument (the functional one). This technique is used to prune incorrect programs from an early stage. All this process is wrapped in a depth-bounded search, so as to ensure the shortest program is found. Our paper has started as an experiment to see whether ideas from *Metagol* could be transferred to a functional setting; hence, in the next chapters we use similar terminology, especially around metarules and background knowledge. However, our implementation treats IBK as a form of metarule. We also use a depth-bounded search in our algorithm, for similar reasons to *Metagol*. Our system does not have

example propagation for now, but we do discuss the pros and cons of having such a pruning mechanism (especially in the context of variable metarules and function reuse) and why such example propagation is very well fitted for ILP, but less so for IFP.

2.2.2 Magic Haskeller

Katayama's *Magic Haskeller* [7] is a GAT approach that uses type pruning and exhaustive search over the program space, using a set of background functions. Katayama argues that type pruning makes the search space manageable. One of the main innovation of the system was the addition of higher-order background functions which speeds up the searching process and helps simplify the output programs. The programs synthesized by the system are chains of function applications. Our system differs in the fact that our programs are modular, each function having the form of a specific metarule, which allows for function reuse. One of *Magic Haskeller*'s limitations is the inability to provide user supplied background knowledge. Our approach enables a user to experiment with the background functions in a programmatic manner, and we also make it fairly easy for a user to change the metarules and hence the grammar of the programs.

2.2.3 λ^2

λ^2 [3] is an IFP system which combines GAT and analytical methods: the search is similar to *Magic Haskeller*, in the way it uses higher order functions and explores the program space using type pruning, but differs in the fact that programs have a nested form (think of *where* clauses in Haskell). However, such an approach does not allow function reuse, since an inner function can not use an "ancestor" function (infinite loop). Our paper tries to address this, exploring the possibility of creating non-nested programs and hence allowing function reuse. Our implementation has a few theoretical guarantees, one of them being the ability to change the metarules without breaking the algorithm. An interesting fact is that λ^2 uses example deduction in IFP, the process of generating new examples based on some deduction rules, similar to how *Metagol* does it. We investigate the effectiveness of example deduction for our implementation, and we go beyond what is presented in λ^2 by investigating example deduction for other combinators (e.g. function composition).

2.3 Function/predicate invention

AC: http://andrewcropper.com/pubs/phd-thesis.pdf (page 19)

2.4 Function reuse

Generally, most IP approaches tend to disregard the extra knowledge found during the synthesis process. In fact, systems like λ^2 and *Magic Haskeller* make this impossible because of how the search is conducted. Some systems, like *Igor 2* do have a limited form of it. This usually stems from what grammars the induced program use. One of our main interests has been the usefulness of function reuse by allowing a modular way of generating programs (that is, we create "standalone" functions that can then be pieced together like a puzzle). For example, consider the *drop lasts* problem: given a non-empty list of lists, remove the last element of the outer list as well as the last elements of all the inner ones. The results of running our system using *reverse* and *tail* as background knowledge can is:

Example 2 (*droplasts* - no function reuse)

```
target = f1.f2
f2 = reverse.f3
f3 = map reverse
f1 = f2.f4
f4 = tail.f5
f5 = map tail
```

However, if function reuse is enabled, not only do we get better computation time, but we also get a simpler program:

Example 3 (*droplasts* - function reuse)

```
target = f1.f2
f2 = map f1
f1 = reverse.f3
f3 = tail.reverse
```

An interesting questions when considering function reuse is what kind of programs benefit from it, which we explore in chapter 6.

Chapter 3

Problem description

In the previous two chapters we have presented inductive programming as a subject and talked about program synthesis in an informal setting. Before presenting the system, we shall give our formalization of the program synthesis problem.

3.1 Abstract description of the problem

A program synthesis system's (PSS) aim is to induce programs with respect to some sort of user provided specification. We call the output of the synthesis process *induced program*, and the functions that are part of it *induced functions*.

Different PSS take various specifications as input. In this paper we consider two types of specification: background knowledge and input-output examples.

Definition 1 (**Background knowledge (BK)**)**.** We define background knowledge to be information used during the synthesis process to invent programs. The BK completely determines the possible forms an induced program can have. More precisely, there are two types of BK:

- *Background functions*: those are either user provided functions or previously induced functions created during the synthesis process.
- *Metarules*: those are a set of rules that describe the forms of the induced functions; they should be seen as templates that contain numbered place-holders ($\boxed{\alpha}_i$) that will eventually be filled with background functions.

Example 4 (Metarules)

Conditional metarule: **if** $\boxed{\alpha}_0$ **then** $\boxed{\alpha}_1$ **else** $\boxed{\alpha}_2$.

Higher-order metarules: $\boxed{\alpha}_0$ **.** $\boxed{\alpha}_0$, **map** $\boxed{\alpha}_0$, **fold** $\boxed{\alpha}_0$, **filter** $\boxed{\alpha}_0$.

Example 5 (Background functions)

```
reverse xs = if xs == []
             then []
             else rev (tail xs) ++ [head xs]

addOne x = x + 1
```

We say an induced function is *complete* if its body contains no place-holders or *incomplete* otherwise.

Definition 2 (**Examples**)**.** Examples are user provided input-output pairs that describes the aim of the induced program. We shall consider two types of examples:

- Positive examples: we use the relation $in \rightarrow^{+} out$ to describe positive examples; this means that an induced program should produce the output out if in is given as the input;
- Negative examples: we use the relation $in \rightarrow^{-} out$ to describe negative examples; this means that an induced program should not produce the output out when in is given as input; one way to look at negative examples is that they try to remove ambiguity: for example, if we are trying to synthesize a program that reverses a list, if the positive examples are only lists of one element then another possible program is the identity function.

Example 6 (Positive and negative examples)
Given the positive examples $[3,2,1] \rightarrow^{+} [1,2,3]$ and $[5,4] \rightarrow^{+} [4,5]$, and the negative examples $[1,3,2] \rightarrow^{-} [2,3,1]$ and $[5,4] \rightarrow^{-} [5,4]$ then the program we want to induce is likely to be a list sorting algorithm. Note that if we only look at the positive examples, another possible induced program is a a list reversing algorithm, but the negative example $[1,3,2] \rightarrow^{-} [2,3,1]$ removes this possibility.

Something to note is that both the positive and the negative examples should have the same structure.

Definition 3 (**Satisfiability**)**.** We say an induced function f satisfies the relations $\rightarrow^{+}$ and $\rightarrow^{-}$ if:

- $\forall (in, out) \in \rightarrow^{+} . f(in) = out$
- $\forall (in, out) \in \rightarrow^{-} . f(in) \neq out$

AC: todo

Definition 4 (**Hypothesis space / program space**)**.**

We can now give the formulation for the program synthesis problem.

Definition 5 (**Induced program**)**.** Given a set of metarules, background functions, positive and negative examples, a program synthesis system should create a set of complete induced functions whose form respects that of the metarules and one of the functions satisfies the positive and negative examples; we call that the *target function*.

AC: a different suggestion below We can now give the formulation for the program synthesis problem.

Definition 6 (**Inductive Haskell problem**)**.** Given:

- a set of positive input/output examples $E^+ = \{(x_i, y_i), \ldots, (x_n, y_n)\}$
- a set of negative input/output examples $E^- = \{(x_i, y_i), \ldots, (x_n, y_n)\}$
- a hypothesis space $\mathcal{H}_{BK,M}$ defined by BK and M

The *Inductive Haskell problem* is to find a hypothesis $H \in \mathcal{H}$ such that:

- $\forall (x, y) \in E^+ \; H(x) = y$
- $\forall (x, y) \in E^- \; H(x) \neq y$

We call H a *solution* to the Inductive Haskell problem.

3.2 Why invent and reuse functions?

AC: motivate it here

Definition 7 (**Hypothesis space size**)**.** Given a set of metarules M and a set of background functions B, and a maximum program size N, the the size of the inductive Haskeller hypothesis space is AC: todo here - look at http://andrewcropper.com/pubs/aaai20-forgetgol.pdf

AC: talk about how reuse would increase the branching factor but decrease the depth e.g. thm1 of http://andrewcropper.com/pubs/mlj19-metaho.pdf

3.3 Properties of the program space

In this section, we assume the set of metarules, the set of background functions, $\rightarrow^+$ and $\rightarrow^-$ to be fixed. When we refer to an induced program, we refer to the induced program given those sets.

Definition 8 (Program space)**.** Given a set of metarules and a set of background functions, we define program space to be the set of all the possible functions that can be induced using those.

Definition 9 (**Specialisation step**)**.** Let f be an incomplete induced function. We call a specialisation step a substitution of the form $f[\beta \leftarrow \boxed{\alpha\_\text{i}}]$, which denotes the fact that the i^{th} place-holder in f is replaced by the expression β. We write this as the relation $f \xrightarrow[\beta]{\boxed{\alpha}_i} f'$, where $f' = f[\beta \leftarrow \boxed{\alpha}_i]$.

The reason we do not allow β to be a metarule if we are substituting a place-holder is that we want each incomplete induced function to become a complete one in a finite amount of specialisation steps. No expressivity is lost here, since a function place-holder can be replaced by another induced function, and that induced function could be a new one (that is, its associated expression is a metarule place-holder).

What else should I include here? Is this the right place for this section, or should i put it in the algorithm chapter?

Chapter 4

Implementation

Since we are trying to find a program that satisfies an instance of the program synthesis problem, it is natural to use a searching algorithm. We will use a depth-bounded search that uses the idea of iterative deepening. This is because we want our programs to gradually increase in size, so as to not have situations where we apply a function infinitely many times before considering other branches (consider a case where we apply the $(map\ \boxed{f}_0)$ metarule again and again). Also, this assures that the induced program will be the simplest one (FORMALIZE SIMPLE). This however does not guarantee anything about the complexity of the induced program: to have some claims about the complexity of the induced program, we should use some other cost function. Intuitively, the nodes in the search tree will be represented by a set of pairs, each describing an induced program. Those pairs will have the form (*function identifier*, *type declaration*, *body*). We call those pairs *function signatures*. Initially this set will be a singleton containing only the function signature $\{(f_t, \theta_f, \boxed{\text{m}})\}$, where f_t is the target function, θ is its type declaration and $\boxed{\text{m}}$ represents the fact that f_t does not yet have a "shape" for its body. The leaves in the search tree will represent a set of complete induced functions that form an induced program. The search will stop once we find a leaf that represents an induced program whose target function satisfies the examples.
We now describe how the searching is done:

1. If the current induced program has no place-holders, test to see whether the target function satisfies the examples.
   - If it does, we have found a complete program that satisfied the examples.
   - If it does not, we undo the last specialisation step and try step 3 again.

2. If the current induced program has place-holders, select one. Suppose we consider the function signature $\{(f, \theta_f, E_f)$, where E_f has at least a place-holder. The specialisation step should be described based on the form of the place-holder:

   – $\boxed{\text{m}}$: then the specialisation step will look like $E_f \xrightarrow[M]{\boxed{\text{m}}} E'_f$, where M is a metarule as described in definition 3.

   – $\boxed{\text{f}}_i$: then the specialisation step will look like $E_f \xrightarrow[\boxed{\text{f}}_i]{\boxed{\text{f}}} E'_f$, where f is either a first-order background function, an existing induced function or a to be added induced function, tried in this order; we expand on what we mean by "to be added": after trying a first-order background function and an existing induced function, we could only go forward by saying that our place-holder will be substituted by another function which we will synthesize in the future; to do this we add a new element to the set of induced functions $(g, \theta_g, \boxed{\text{m}})$.

Chapter 5

Testing the system

Chapter 6

Conclusions

6.0.1 Summary

6.0.2 Reflections

6.0.3 Limitations

6.0.4 Extension of the project

Bibliography

[1] Andrew Cropper. *Efficiently learning efficient programs*. PhD thesis, Imperial College London, 2017.

[2] Andrew Cropper, Rolf Morel, and Stephen H. Muggleton. Learning higher-order logic programs. 2019.

[3] John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50:229–239, 06 2015.

[4] Sumit Gulwani. Example-based learning in computer-aided stem education. *Communications of the ACM*, 57(8):70–80, August 2014.

[5] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. volume 55, pages 97–105, January 2012. Invited to CACM Research Highlights.

[6] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):9099, October 2015.

[7] Susumu Katayama, 2005.

[8] Martin Kato, Emanuel Kitzelmann, and Ute Schmid. A unifying framework for analysis and evaluation of inductive programming systems. *Proceedings of the 2nd Conference on Artificial General Intelligence, AGI 2009*, 06 2009.

[9] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. 2009.

[10] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2:90–121, 01 1980.

[11] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3), Dec 1995.

[12] Stephen Muggleton. Inductive logic programming: Issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114(1):283 – 296, 1999.

[13] Ute Schmid. Inductive programming as approach to comprehensible machine learning. In *DKB/KIK@KI*, 2018.

[14] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161175, January 1977.