On inductive synthesis of declarative programs

Bjarte M. Østvold
Department of Computer and Information Science
Norwegian University of Science and Technology
Gjøsåhagen, N-7034 Trondheim, Norway
Phone: +47 73 59 44 83; fax +47 73 59 44 66
Email: bjartem@idt.ntnu.no

Abstract

We consider how different kinds of and aspects of declarative programming languages — logic, functional, type system — make them suitable for use in the context of inductive synthesis of programs. Then we formulate the inductive synthesis problem for functional programs and outline an algorithm for solving it. We conclude that higher-order definitions and type systems are very useful in inductive synthesis.

1 Introduction

Deductive reasoning techniques are well established in program synthesis and transformation [15, 4]. They are concerned with making a complete specification operational or making some initial program more efficient without changing its semantics. There has been far less interest in inductive (unsound) reasoning techniques in program synthesis; Smith surveys some early work [17]. With the recent interest in inductive logic programming (ILP) [14] machine learning researchers have investigated the use of induction in logic program synthesis. The goal is to (partially) automate the construction of programs implementing (recursive) algorithms by unsound reasoning from an incomplete specification in the form of examples and general problem-independent background knowledge. Drawing on the insights of these researchers [8; 6; 7; 1; 2] we identify the following problems for such synthesis:

- The induced program may have a different meaning compared with what we intended — this problem is inherent to this synthesis approach, but it is also its strength: we need not have or specify completely predicate we are synthesising.
- The search space is vast — we need somehow to constrain search by limiting the number of programs considered.
- Specification by examples only is too weak a setting for all but trivial synthesis problems. Also, specification by spoon-feeding — requiring many or particularly well-chosen examples — is clearly undesirable. And negative examples, being non-instances of the solution, can be very awkward to provide.
- Human programmers have a lot of knowledge about programs and programming that must be represented and made available to a synthesis algorithm. Important examples are algorithm schemas, e.g., 'divide and conquer', and knowledge about concrete programs, like types and modes.

We consider how these problems can be addressed in the following.

2 Declarative programming languages and inductive synthesis

Due to their clean semantics — at least for language subsets — and their meta-programming facilities declarative programming languages are suitable both for representing programs and knowledge about programs, and for implementing synthesis algorithms. In this section we consider how different kinds of and aspects of declarative programming languages make them suitable for use in the context of inductive synthesis.

In current work on inductive synthesis Prolog dominates almost completely and in the related area of genetic programming [11] LISP is the language of choice. Languages with a weak type system has the advantage that efficient meta-programming is easier to provide for language implementors, but this advantage is not inherent to weakly typed languages and statically typed languages are catching up [9; 5]. Static type systems have the advantage that they allow us to reject many programs statically, i.e., without testing them. Prolog’s
-- divide and conquer schema
dac :: (a -> Bool, a -> b, -- Haskell is curried
      a -> (c, a), c -> b -> b) -> a -> b

dac ops @(baseCase, solve, decompose, compose) x =
  if baseCase x then
    solve x
  else
    let (y, y') = decompose x in
    compose y (dac ops y')

-- list append
append :: [a] -> [a] -> [a]
append xs ys =
  dac (null,
       \_ -> ys, -- lambda function
       \as -> (head as, tail as),
       (:) ) xs

-- n!
fac :: Int -> Int
fac n = dac ( (0=0),
              \_ -> 1,
              \m -> (m, (\k -> k - 1) m),
              (*) ) n

Figure 1: Simple divide and conquer schema and its usage.

mode language [13] is a step in this direction; but it stops short of proving polymorphism.

Definitions of logic predicates are frequently written in a way making them more flexible than definitions in
functional programming, e.g., the predicate append/3 can be used "backwards" to split a list in two. For a
predicate to work in several modes generally requires that its definition is more complex, and thus harder to
learn. Functional programmers on the other hand are happy to write only definitions with a single mode and
returning just one answer. Thus logic predicates as target concepts should only be used when the added flexi-

There appears to be agreement that schemas, sometimes called templates or skeletons, are essential in
inductive synthesis as they capture recurring patterns in algorithm design [6; 2]. An advantage of functional pro-
gramming is that schemas can be represented as first class values using higher-order function definitions, and
thus staying within the type system with the advantages discussed above. Fig. 1 shows a simple divide and con-
quer schema written in Haskell [16] — a statically typed functional programming language — and examples of its
usage. Haskell and other functional languages have pro-

-- higher order definition for lists
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = x 'f' (foldr f a xs)

-- other definitions using 'foldr'
append :: [a] -> [a] -> [a]
append xs ys = (foldr (:)) ys xs

length :: [a] -> Int
length xs = foldr (\_, n -> succ n) 0 xs

map :: (a -> b) -> [a] -> [b]
map f xs = foldr ((:) . f) [] xs

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x xs ->
                   if p x then x:xs else xs) []

Figure 2: Higher-order definition for lists and its usage.

defined higher-order definitions that can be used to
give short and elegant definitions of other functions, and
short definitions means less search. Some examples are
given in Fig. 2. There is a systematic way to construct
powerful higher-order function definitions for particular
algebraic datatypes [12].

Logic programs define relations and relations have a
major advantage over functions: there is a most gen-
eral relation for a domain, whereas there is no most
general function for any reasonable definition of 'general'.
Generality orders form the basis for search in
IPL algorithms (top-down, bottom-up), and in addition
generalisation/specialisation of logic programs can be
characterised syntactically — add/delete a literal in a
programs clause or apply substitution/inverse substitu-
tion to it — thus making search operators efficient and
easy to implement. (More fine-grained orders of gener-
ality exist, but they require more complex operators [18;
13].) For inductive synthesis of functional programs the
lack of generality orders is a serious problem. An alter-
native approach for such synthesis is outlined in the next
sections.

3 Inductive synthesis of functional programs

Here we formulate a version of the inductive synthesis
problem for functional programs, give a naive enumera-
tion algorithm for solving it, and indicate how the algo-

3.1 Problem formulation

A simple formulation of the inductive synthesis problem
for functional programs (extensions are discussed in §4):

1 Prolog is not a system tailored program synthesis.

2 No doubt this claim will be contested by die-hard logic

programmers!
Given a library $L$ of function definitions, a type signature $\tau$ of an unknown function $f$ and examples of equations $E$ involving $f$, 
\[ \{ f(a_1, \ldots, a_m) = b_1, \ldots, f(a_n, \ldots, a_{nm}) = b_n \}. \]
Find a definition $\text{def}(f)$ for $f$ such that $\text{def}(f)$ has type $\tau$, is built using only definitions from $L$ and recursion, and reduce all left-hand sides in $E$ to right-hand sides in a finite number of steps.

Some comment on the problem formulation:

- For simplicity we will assume that the $a_i$'s and $b_i$'s do not involve $f$.
- We are not interested in function definitions that ignore their arguments completely.

Comparing with a typical inductive synthesis problem for logic programs using ILP we note the following:

- The type of the unknown function is available.
- There are no negative examples, but given that we are learning a function they could be constructed trivially. However, we shall see later that they are of limited interest.
- The solution is not more general than the examples $E$ in a logical sense, but in an operational sense — proving equations by reduction.

### 3.2 Naive algorithm

In the following, we assume that our candidate solutions or hypotheses, as well as the library $L$, are definitions written in Haskell.

As a first attempt we use an enumeration algorithm. Viewing expressions as trees we can define an order on expressions based on the number of symbol occurrences, i.e., we order expressions based on syntactic size. Example: synthesising list append with list operations in $L$. Problem specification:

```haskell
append :: [a] -> [a] -> [a] -- tau
append [1,2] [3,4] == [1,2,3,4] -- E
```

- library $L$ defines:
  - [], ::, head, tail --- standard list primitives

We assume that recursion is on a reduced version of the first argument — this is a heuristic decision, and in a more complex case we may have to try several alternatives. Considering the base case the left-hand side of the definition is $\text{append} \, [] \, ys$. Candidate right-hand sides: $[]$, $ys$, $\text{head} \, [], \text{head} \, ys$, tail $[]$, tail $ys$, ...

The enumeration algorithm consists in testing expressions of increasing syntactic size on the examples until a valid definition of $f$ is found (we also need a recursive case for this, see below). However, that many expressions can be rejected without testing on the example since:

- $\text{def}(f)$ must have the correct type;
- $\text{def}(f) \neq \bot$, e.g., not head $[]$;
- $\text{def}(f)$ must use (part of) each argument $a_i$ in $E$;
- recursion must be sound;

In this case, the algorithm only need to test four hypotheses to find a solution: combinations of one base case alternative and four recursive case alternatives:

```haskell
-- simplest base case
append [] xs = xs
```

```haskell
-- recursive cases of increasing or equal complexity
append (x:xs) ys = append xs ys
append (x:xs) ys = x:(append xs ys)
```

The synthesis can proceed from any single example that distinguishes these four hypotheses including the one given above. From this we draw the conclusion that synthesising a functional version of $\text{append}$ is a trivial problem. Its relational counterpart is frequently used as an example in ILP papers, but few if any ILP systems can learn it from one example. This suggests both that learning predicates is harder than learning functions and that most ILP algorithms are not tailored to synthesis of recursive definitions.

But: enumerative search leads immediately to a combinatorial explosion for more complex problems.

### 3.3 Improving the algorithm

In this section we discuss some ways of improving on the enumeration search strategy.

First we note that the space of expressions contains a number of equivalent expressions that gives rise to redundant search. Some examples of such expression equivalences are shown in Fig. 3. The algorithm can safely eliminate the expressions on the left-hand sides. Determining equivalent pairs of expressions is non-trivial in general as it involves proving equivalences. Since reduction preserves equivalence a subclass of redundant expressions are those that reduce to smaller ones — smaller expressions precede the original ones in the order — and some such cases can be discovered by performing a few reduction steps and comparing sizes. The first example in the figure comes in this category: the left hand side reduces to the right hand side in one step. Other examples, such as associativity of addition on integers — an algebraic theorem — cannot be dealt with in this way.

Second, one can make the assumption that the solution definitions are always recursive — a safe assumption, and additionally assume that recursion only occur
indirectly through the use of recursive higher-order function definitions [3, 12] or schemas. By eliminating direct recursion and having a minimal set of schemas the search space the algorithm must consider is made smaller still. Once a solution is obtained a deductive synthesis technique such as partial evaluation [10] could be used to introduce direct recursion. Finally, there is an important difference between our algorithm as described so far and ILP algorithms: we do ‘big-bang learning’, only using the examples E as a validation test to accept or reject complete hypotheses, while in ILP testing a hypothesis on examples can lead to a refined hypothesis, e.g., specializing a clause to avoid covering a negative example in top-down search. However, as the following example will show, we can partly get back to such a situation. Example: synthesizing mapList. Problem specification:

mapList :: (a -> b) -> [a] -> [b] -- tau
mapList (1+) [1,2,3] == [2,3,4] -- E

-- library L defines:
[.], : -- list constructors
. -- functional composition
foldr -- only schema

By assumption we must use foldr — the only schema available. We assume that recursion is on the list argument — again this is heuristic. Initial hypothesis:

mapList u xs = foldr k a xs

Now, compared to the previous section our problem is reduced from finding a single recursive function for mapList to finding two smaller non-recursive functions k and a, letting the schema foldr take care of the recursion. Based on E and the initial hypothesis we know that the following equation holds:

foldr k a [1,2,3] == [2,3,4]

Unfolding foldr repeatedly gives ("k" is the infix form of k):

1 'k' (2 'k' (3 'k' a)) == [2,3,4]  

Here we must solve a subproblem: finding k. Type of and example for k are obtained from the above equation and the problem specification for mapList:

k :: Int -> [Int] -> [Int]
1 'k' <something> == [2,3,4]  

We know that either k or a have to use the argument u to mapList — (1+) in the example, the increment function. Enumerating possibilities for k gives as the first well-typed expression using u:

k = (:) , (1+)  

Going back to the original problem we get by insertion for K and simplification:

mapList u xs = foldr ((:) . u) [] xs

This gives a solution for a without search:

a == []  

Complete solution definition:

mapList u xs = foldr ((:) . u) [] xs

Note that negative examples would be of little use given the constructive way in which E is used above; negative examples could, however, be used to test for validity. The above example synthesis just illustrate the idea: a number of issues need to be investigated before a complete algorithm can be made.

4 Concluding remarks

We sum up our conclusions as follows:

• type systems help restrict the search space;

• datatypes and higher-order constructs are useful since they embody algorithm design knowledge and lead to succinct formulation of solutions, and thus restricts search;

• inductive synthesis of functional programs is interesting and some initial progress has been made.

Future work on inductive synthesis of functional programs includes:

• Allow the problem formulation to specify intermediate data structure solutions must use, and possibly new types and operations to be used on the data structures.

• Although not strictly necessary, heuristics for finding a modular solutions comprised of several definitions should be a useful search heuristic — after all human programmers do this.

• Since in practice we have to cut off reduction sequences after a limited number of steps a correct, but inefficient definition could be rejected and search led astray. We need to find schema sets leading to reasonable efficient hypotheses.
• We should investigate ways of taking advantage of known properties of definitions in \( L \) during search, e.g., inverses or axioms of datatypes. We note that the integers with axioms and theorems are not in principal different from other algebraic datatype, and thus pose no special difficulties. This cannot be said for logic programming.

• Improve the current implementation based on enumeration to include the ideas in §3.

Acknowledgements: This research was initiated while the author visited Department of Computer Science, University of Bristol, U.K. Particularly useful were discussions with John Lloyd, Christophe Giraud-Carrier and Tony Bowers on program learning and with Chris Dornan and Ian Holeyer on functional programming.

References


