Data-Driven Induction of Recursive Functions from Input/Output-Examples

Emanuel Kitzelmann

Faculty of Information Systems and Applied Computer Sciences
University of Bamberg

ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming

Outline

1. Inductive (Functional) Programming
   - General Approaches
   - Formalization
   - Analytical Function Induction
2. Function Induction by Pattern Refinement, Matching, and Ubiquitous Subprogram Introduction
   - General Algorithm
   - Processing Unfinished Rules
3. Experimental Results

Outline

1. Inductive (Functional) Programming
   - General Approaches
   - Formalization
   - Analytical Function Induction

2. Function Induction by Pattern Refinement, Matching, and Ubiquitous Subprogram Introduction
   - General Algorithm
   - Processing Unfinished Rules

3. Experimental Results

General IP Approaches

	Search-Based IP (generate-and-test)	*Analytical IP (data-driven)*
general method	*programs are generated and then tested against given examples*	*programs are derived from given examples*
program classes	*unrestricted*	*restricted*
usage of predefined functions	*possible*	*not possible*
induction times	*high*	*short*
possible application fields	*inventing new and efficient algorithms*	*programming assistence, enduser programming*
prototypical systems	ADATE	DIALOGS, IGOR I

Our Contribution

An IP algorithm that

- combines the analytical approach with search to achieve both relatively unrestricted inducible program classes and reasonable induction costs,
- deals with arbitrary user-defined algebraic datatypes,
- automatically “invents” necessary subprograms,
- facilitates use of background knowledge,
- induces programs for multiple interdependent target functions,
- induces rather complex recursive patterns (nested calls of induced recursive functions, mutual recursion, tree recursion, arbitrary numbers of base- and recursive cases).

Outline

1. Inductive (Functional) Programming
   - General Approaches
   - Formalization
   - Analytical Function Induction
2. Function Induction by Pattern Refinement, Matching, and Ubiquitous Subprogram Introduction
   - General Algorithm
   - Processing Unfinished Rules
3. Experimental Results

Representation Language – Example

Hypotheses (induced programs), background knowledge and examples are represented as *equations* fulfilling some structural conditions over typed first-order signatures.

Example (A List Reversing Program)

Signature: $\Sigma = \mathcal{D} \cup \mathcal{C}$, $\mathcal{D} = \{Reverse, Last, Init\}$, $\mathcal{C} = \{[], cons\}$.
Equations:

$$
\begin{aligned}
Reverse([]) &= [] \\
Reverse([X|Xs]) &= [Last([X|Xs])|Reverse(Init([X|Xs]))] \\
Last([X]) &= X \\
Last([X_1, X_2|Xs]) &= Last([X_2|Xs]) \\
Init([X]) &= [] \\
Init([X_1, X_2|Xs]) &= [X_1|Init([X_2|Xs])]
\end{aligned}
$$

Representation Language

- hypotheses, background knowledge and examples are *equations* of a particular form over typed first-order signatures Σ
- Σ is partitioned into a set of *defined function symbols* $\mathcal{D}$ and a set of *constructors* $\mathcal{C}$
- equation left hand sides (lhss) have the form

$$F(t_1, \ldots, t_n)$$

  where F is a defined function symbol and the t_i are composed of constructors and variables
- the argument list $t_1, \ldots, t_n$ is called *pattern*
- the described form of equations corresponds to the concept of *pattern matching* in functional languages

Term Rewriting as Operational Semantics

- equations can be read from left to right as rewriting (simplification) rules $\rightsquigarrow$ *term rewriting system (TRS)*; a TRS whose rules have the “pattern matching” form are called *constructor systems (CSs)*
- the rewrite relation induced by a TRS is denoted by $\rightarrow_R$, its reflexive transitive closure by $\stackrel{*}{\rightarrow}_R$
- a term that cannot be further simplified is called *normal form*; if $t \stackrel{*}{\rightarrow} s$ and s is in normal form we write $t \stackrel{!}{\rightarrow} s$ and say that t normalizes to s or that s is a normal form of t
- required characteristics: *termination* and *confluence*

Characterizing Hypotheses and Induction Algorithms

Definition (Correctness of Hypotheses)

A hypothesis, i.e., a CS R, is *correct* with respect to a set of example equations iff $F(i) \stackrel{!}{\rightarrow}_R o$ for each example equation $F(i) = o$.

Definition (Soundness and Completeness of IP Algorithms)

- An induction algorithms is *sound* iff it induces only correct hypotheses.
- An induction algorithm is *complete* with respect to a hypothesis space $\mathcal{H}$ iff it finds a hypothesis if there exists a correct hypothesis in $\mathcal{H}$.

The Inductive Functional Programming Problem

Given

- a hypothesis space $\mathcal{H}$,
- a set of example equations E and
- some background knowledge B,

find a hypothesis (set of equations) $H \in \mathcal{H}$ such that $H \cup B$

- constitutes a terminating and confluent constructor system that is correct with respect to E and
- computes arbitrary complex inputs.

Inductive Functional Programming Example

Examples for Target Function *Reverse* and background knowledge *Last*

$$
\begin{aligned}
Reverse([]) &= [] \\
Reverse([a]) &= [a] \\
Reverse([a,b]) &= [b,a] \\
Reverse([a,b,c]) &= [c,b,a] \\
Reverse([a,b,c,d]) &= [d,c,b,a]
\end{aligned}
\qquad
\begin{aligned}
Last([a]) &= a \\
Last([a,b]) &= b \\
Last([a,b,c]) &= c \\
Last([a,b,c,d]) &= d
\end{aligned}
$$

Induced Constructor System

$$
\begin{aligned}
&Reverse([]) && \rightarrow [] \\
&Reverse([X|Xs]) && \rightarrow [Last([X|Xs])|Reverse(Init([X|Xs]))] \\
&Init([X]) && \rightarrow [] \\
&Init([X_1, X_2|Xs]) && \rightarrow [X_1|Init([X_2|Xs])]
\end{aligned}
$$

Outline

Analytical Function Induction

Theorem (Analytical Function Induction)

Let E be a set of example equations, p be a pattern, and E_p be the subset of E for which the inputs match p with substitutions σ. Let $C[]$ be a context[a] and r be a term. Then

$$(E \setminus E_p) \;\cup\; \{\, F(p) = C[F(r)] \,\}$$

is correct with respect to E if for all examples $(F(i) = o) \in E_p$ (with $i = p\sigma$) exist an example $(F(i') = o') \in E$ such that:

$$o = C\sigma[o'] \;\wedge\; i' = r\sigma$$

[a] a term containing a distinguished symbol $\square$, called wole, at any position; then $C[s]$ denotes the result of replacing the whole by s in $C[]$

Analytical Function Induction Example

Let $p = [X|Xs]$ be a pattern and E be the following example equations:

$$\begin{array}{l} 1.\ Unpack([]) = [] \\ \left.\begin{array}{l} 2.\ Unpack([b]) = [[b]] \\ 3.\ Unpack([a,b]) = [[a],[b]] \end{array}\right\} E_p \end{array}$$

Then holds:

$$\begin{array}{lll} o_2 = [[X]|o_1]\sigma_2 & i_1 = Xs\sigma_2 & \text{with } \sigma_2 = \{X \leftarrow b, Xs \leftarrow []\} \\ o_3 = [[X]|o_2]\sigma_3 & i_2 = Xs\sigma_3 & \text{with } \sigma_3 = \{X \leftarrow a, Xs \leftarrow [b]\} \end{array}$$

And thus:

$$\begin{array}{l} Unpack([]) = [] \\ Unpack([X|Xs]) = [[X]|Unpack(Xs)] \end{array}$$

is correct with respect to E.

Outline

Inductive Bias

A correct hypothesis is chosen according to the following conditions:

most specific patterns
: each pattern is the *least general generalization (lgg)* of all matching example inputs (narrows pattern search space)

non-unifying patterns
: no two patterns unify, i.e., each example input matches *exactly one* pattern (assures confluence)

minimal number of case distinctions
: number of example input subsets induced by the patterns is minimal, i.e., no correct CS with fewer “case distinctions”

General Algorithm

- kind of best first search in the space of complete and terminating CSs with most specific, non-unifying patterns
- initial state: with respect to the “minimal number of case distinctions” criterion – one initial rule per target function
- during search, rhss may contain variables not occuring in lhss, such rhss and rules are called *unfinished*
- unfinished rules of currently best hypotheses are replaced by (sets of) successor rules repeatedly until goal state is reached
- goal states: at least one of the currently best hypotheses is finished (i.e., contains no unfinished rule)
- same rule may be member of different hypotheses, hence several hypotheses are processed in parallel

Initial Rules

- given a set of example equations, the *initial rule* contains the lgg of the example inputs as pattern and the lgg—with respect to substitutions for input lgg—of the example outputs as rhs
- only initial rules are (possibly) unfinished, i.e., processing an unfinished rule either finishes the rule or completely replaces the rule by a *set* of new initial rules

Example

Example equations:

$$Unpack([b]) = [[b]], \quad Unpack([a,b]) = [[a],[b]]$$

Initial rule:

$$Unpack([X|Xs]) = [[X]|Y]$$

Symbols in RHSs and Processing Unfinished RHSs

- symbols in finished RHSs: constructors, pattern variables, defined function symbols
- symbols in unfinished RHSs: additionally “unbound” variables (to be “removed”) but no defined function symbols
- thus: to finish a RHS, subterms at positions on pathes from the root to “unbound” variables has to be replaced by calls to defined functions

Example

Example equations:

$$Unpack([b]) = [[b]], \quad Unpack([a, b]) = [[a], [b]]$$

Initial rule:

$$Unpack([X|Xs]) = [[X]|Y]$$

Outline

Splitting Rules by Pattern Refinement

Introducing a Case Distinction

Replaces an unfinished rule with pattern p by at least two new rules with more specific patterns in order to establish a case distinction.

Method

1. chose a position u with a variable in p
2. take all example inputs with *same* constructor at position u into same subset
   $\rightsquigarrow$ partitions examples
3. compute initial rules for all new example subsets

Splitting Rules – Example

example equations:

1. $Unpack([a]) = [[a]]$
2. $Unpack([b]) = [[b]]$
3. $Unpack([b, c]) = [[b], [c]]$
4. $Unpack([a, b, c]) = [[a], [b], [c]]$

current unfinished rule: $Unpack([X|Xs]) = [[X]|Y]$

subsets for position 2 (variable *Xs*):

$\{1, 2\}$ with same constructor []

$\{3, 4\}$ with same constructor *cons*

new initial rules:

$$Unpack([X]) = [[X]]$$
$$Unpack([X_1, X_2|Xs]) = [[X_1], [X_2]|Y]$$

Introducing (Recursive) Function Calls

Applying the Analytical Function Induction Principle

Replaces the rhs of an unfinished rule F with pattern p by a (recursive) call to the/another target function or background knowledge function F'.

Method

1. for a fixed defined function F' find an example equation $F'(i') = o$ for each (current) example equation $F(i) = o$
2. introduce a new defined function symbol R and example equations $R(i) = i'$
3. replace the unfinished rhs by $F'(R(p))$; this finishes the rule
4. induce R from its examples

Introducing Function Calls – Example

all given example equations E for *Last*:

$$\begin{array}{l} 1.\ Last([b]) = b \\ 2.\ Last([c]) = c \\ \left.\begin{array}{l} 3.\ Last([a,b]) = b \\ 4.\ Last([b,c]) = c \\ 5.\ Last([a,b,c]) = c \end{array}\right\} E_{[X_1,X_2|Xs]} \end{array}$$

current unfinished rule: $Last([X_1,X_2|Xs]) = Y$

new examples:

$$R([a,b]) = [b], \quad R([b,c]) = [c], \quad R([a,b,c]) = [b,c]$$

finished rule: $Last([X_1,X_2|Xs]) = Last(R([X_1,X_2|Xs]))$

Introducing Subfunctions For Unfinished Subterms

Dealing with *Unfinished Subterms* as New Induction Problems

Did you miss (recursive) function calls at "deeper" positions in the rhs, i.e., within a context $C[]$ *as described for the analytical principle?*

Here you are!

Introducing Subfunctions For Unfinished Subterms

Dealing with *Unfinished Subterms* as New Induction Problems

Replaces unfinished subterms of an unfinished rhs for a rule with pattern p by calls to new subfunctions. Precondition: Root of rhs is a constructor c.

Method

1. replace each direct unfinished subterm of c by a call to a new subfunction $Sub(p)$; this finished the rule
2. induce each new subfunction Sub from example equations $Sub(i) = o|_k$ where k is the index of the subterm replaced by $Sub(p)$

Introducing Subfunctions – Example

example equations:

$$Unpack([b]) = [[b]], \quad Unpack([a,b]) = [[a],[b]]$$

current unfinished rule:

$$Unpack([X|Xs]) = [[X]|Y]$$

finished rule:

$$Unpack([X|Xs]) = [[X]|Sub([X|Xs])$$

example equations for *Sub*:

$$Sub([b]) = [], \quad Sub([a,b]) = [[b]]$$

Implementation

- prototype of the described algorithm implemented in the reflective, term rewriting language MAUDE.
- two extensions:
  1. example equations may contain variables such that necessary amount of example equations decreases
     - $\rightsquigarrow$ advantageous for the user
     - $\rightsquigarrow$ smaller induction times
  2. pattern variables can be tested for equality; establishes a second form of case distinction

Empirical Results

on a P4 with Linux and the MAUDE 2.3 interpreter

- *Length*, *Last*, *Reverse*, *Member*, *Take* (keeps first *n* elements and "deletes" the rest), *Insertion Sort* (with *Insert* as background knowledge) *Even*, *Odd*, *Add*, *Mirror* (mirrors a binary tree) induced in milliseconds from $\leq$ 6 examples (*Member* 13 examples)
- *Reverse* has been specified in two variants:
  1. without background knowledge; *Last* and *Init* automatically introduced
  2. with *Snoc* (inserts an element at the end of a list) as background knowledge
- *Quicksort* with *Append* and the partitioning functions as background knowledge induced in 63 seconds; but depends on applied reduction order, can be induced within one second

Conclusion

- “pure” analytical approaches too restrictive, enumerative “pure” generate-and-test approaches too expensive, so try a combination
- that is: apply a search but use data-driven successor functions
- we have developed such an algorithm and implemented it; the algorithm is complete and sound
- our algorithm is more powerfull than existing analytical approaches to inductive programming and on some tested problems more time efficient than existing generate-and-test base approaches
- future research must try to increase time efficiency