Analytical Inductive Functional Programming

Emanuel Kitzelmann

Cognitive Systems Group
University of Bamberg

International Symposium on
Logic-Based Program Synthesis and Transformation

Outline

1. Inductive (Functional) Programming
   - Introduction
2. Our Approach
   - Basic Concepts
   - The IGOR2 Algorithm
   - Empirical Results and Further Research

Outline

1. Inductive (Functional) Programming
   - Introduction
2. Our Approach
   - Basic Concepts
   - The IGOR2 Algorithm
   - Empirical Results and Further Research

Inductive (Functional) Programming

- constructing programs (*functional* programs here) from input/output-examples (I/O-examples) or other kinds of *incomplete specifications*
- also known as *inductive program synthesis*

Example (*Reverse*)

I/O-examples (containing variables):

$$Reverse([\,]) = [\,],\ Reverse([X]) = [X],\ Reverse([X, Y]) = [Y, X],\ \dots$$

Induced program:

$$\begin{aligned} Reverse([\,]) &= [\,] \\ Reverse([X|Xs]) &= Reverse(Xs)\ @\ [X] \end{aligned}$$

Two General Approaches

Generate-and-test: (heuristically) enumerate and test programs until one is found which correctly computes all I/O-examples

systems: ADATE(Olsson), MAGICHASKELLER(Katayama), FOIL(Quinlan)

Analytical: detect syntactical regularities, imposed by (hypothetical) repeated recursive calls, within and between I/O-examples or traces and derive a recursive function definition from them

systems: THESYS(Summers), IGOR1(Kitzelmann & Schmid), CRUSTACEAN (Aha et al.)

Characteristics of the Two Approaches and Our Idea

The analytical approach:

+ no search in program space and no evaluation of I/O-examples, thus fast
– strongly restricted program schemas
– no background knowledge

The enumerative approach:

+ theoretically no restrictions
– practically very expensive (combinatorial explosion and repeated evaluation of I/O-examples)

Our idea: combine both approaches:

- search in a relatively unrestricted program space, but
- compute successor programs based on regularities between I/O-examples

Characteristics of the Two Approaches and Our Idea

The analytical approach:

- $+$ no search in program space and no evaluation of I/O-examples, thus fast
- $-$ strongly restricted program schemas
- $-$ no background knowledge

The enumerative approach:

- $+$ theoretically no restrictions
- $-$ practically very expensive (combinatorial explosion and repeated evaluation of I/O-examples)

Our idea: combine both approaches:

- search in a relatively unrestricted program space, but
- compute successor programs based on regularities between I/O-examples

Outline

1. Inductive (Functional) Programming
   - Introduction
2. Our Approach
   - Basic Concepts
   - The IGOR2 Algorithm
   - Empirical Results and Further Research

Basic Concepts

- functional programs are *sets of equations over typed*, user-defined, *signatures*
- evaluation by reading them as *rewrite rules*
- equations/rules form a *constructor (term rewriting) system (CS)*, i.e.
  - each function symbol denotes either a (type) *constructor* or a *defined function* (defined by equations)
  - rules have the form $F(p_1, \ldots, p_k) = t$ where F is a defined function symbol and the p_i, called *pattern*, are *constructor terms* (terms consisting of constructors and variables only)
- defined function symbols denote either *target functions* (to be induced) or *background knowledge* (functions assumed to be already implemented)
- I/O-examples (example equations) are equations/constructor systems with constructor terms as rhss
- we have example equations both for target functions and background knowledge

Basic Concepts

- functional programs are *sets of equations over typed*, user-defined, *signatures*
- evaluation by reading them as *rewrite rules*
- equations/rules form a *constructor (term rewriting) system (CS)*, i.e.
  - each function symbol denotes either a (type) *constructor* or a *defined function* (defined by equations)
  - rules have the form $F(p_1, \ldots, p_k) = t$ where F is a defined function symbol and the p_i, called *pattern*, are *constructor terms* (terms consisting of constructors and variables only)
- defined function symbols denote either *target functions* (to be induced) or *background knowledge* (functions assumed to be already implemented)
- I/O-examples (example equations) are equations/constructor systems with constructor terms as rhss
- we have example equations both for target functions and background knowledge

Basic Concepts

- functional programs are *sets of equations over typed*, user-defined, *signatures*
- evaluation by reading them as *rewrite rules*
- equations/rules form a *constructor (term rewriting) system (CS)*, i.e.
  - each function symbol denotes either a (type) *constructor* or a *defined function* (defined by equations)
  - rules have the form $F(p_1, \ldots, p_k) = t$ where F is a defined function symbol and the p_i, called *pattern*, are *constructor terms* (terms consisting of constructors and variables only)
- defined function symbols denote either *target functions* (to be induced) or *background knowledge* (functions assumed to be already implemented)
- I/O-examples (example equations) are equations/constructor systems with constructor terms as rhss
- we have example equations both for target functions and background knowledge

Basic Concepts

- functional programs are *sets of equations over typed*, user-defined, *signatures*
- evaluation by reading them as *rewrite rules*
- equations/rules form a *constructor (term rewriting) system (CS)*, i.e.
  - each function symbol denotes either a (type) *constructor* or a *defined function* (defined by equations)
  - rules have the form $F(p_1, \ldots, p_k) = t$ where F is a defined function symbol and the p_i, called *pattern*, are *constructor terms* (terms consisting of constructors and variables only)
- defined function symbols denote either *target functions* (to be induced) or *background knowledge* (functions assumed to be already implemented)
- I/O-examples (example equations) are equations/constructor systems with constructor terms as rhss
- we have example equations both for target functions and background knowledge

Analytical Function Induction

Theorem

Let E be a set of example equations, F a defined function symbol occurring in E, p a pattern for F, and $E_{F,p} \subseteq E$ the example equations for F whose inputs match p with substitutions σ.

Let C be a context, F', F'' (further) defined function symbols occuring in E.

If for all $F(i) = o \in E_{F,p}$ exist equations $F'(i') = o' \in E$ such that

$$o = C\sigma[o'] \quad \text{and} \quad F''(i) = i'$$

then

$$(E \setminus E_{F,p}) \cup \{F(p) = C[F'(F''(p))]\} \models E$$

Outline

1 Inductive (Functional) Programming
- Introduction

2 Our Approach
- Basic Concepts
- The IGOR2 Algorithm
- Empirical Results and Further Research

Characteristics

Given example equations for target functions E and background knowledge B, IGOR2 returns a set of equations P (a *hypothesis*), constituting a confluent CS, such that

$$F(\boldsymbol{i}) \overset{!}{\rightarrow}_{P \cup B} o \quad \text{for all } F(\boldsymbol{i}) = o \in E$$

Inductive bias

- fewest number of case distinctions (fewest number of different patterns)
- patterns pairwise non-unifying (to guarantee confluence)
- patterns are *least general generalizations* (lggs) of the example inputs they respectively subsume

During search, hypotheses are *unfinished*, meaning that they contain variables in rhss not occuring in lhss.

The General Algorithm

The general algorithm is a kind of *best first search* in the program space.

IGOR2

$h \leftarrow$ the initial hypothesis (one rule per target function)
$H \leftarrow \{h\}$
while h unfinished **do**

- $r \leftarrow$ an unfinished rule of h
- $S \leftarrow$ all successor rule sets of r
- **foreach** $h \in H$ with $r \in h$ **do**
    - remove h from H
    - **foreach** successor set $s \in S$ **do**
        - add h with r replaced by s to H
- $h \leftarrow$ a hypothesis from H with least number of case distinctions

return h

Initial Rules

- given a set of example equations for one target function, the **initial rule** is the **least general generalization (lgg)** of the example equations
- only initial rules are (possibly) unfinished, i.e., processing an unfinished rule either finishes the rule or completely replaces it by a set of new initial rules

Example

Example equations:

$$Reverse([X]) = [X], \quad Reverse([X, Y]) = [Y, X]$$

Initial rule:

$$Reverse([X|Xs]) = [Z|Zs]$$

Initial Rules

- given a set of example equations for one target function, the **initial rule** is the **least general generalization (lgg)** of the example equations
- only initial rules are (possibly) unfinished, i.e., processing an unfinished rule either finishes the rule or completely replaces it by a set of new initial rules

Example

Example equations:

$$\mathit{Reverse}([X]) = [X], \quad \mathit{Reverse}([X, Y]) = [Y, X]$$

Initial rule:

$$\mathit{Reverse}([X|Xs]) = [Z|Zs]$$

Processing Unfinished Rules

Example (Initial Rule)

$$Reverse([X|Xs]) = [Z|Zs]$$

Problem: unbound variables Y, Ys in rhs, i.e., unfinished

Three possible solutions (successor functions):

1. **partition** the example equations and compute initial rules for each subset
2. treat subterms of the rhs containing unbound variables as **new (sub)problems**
3. replace rhs by a **defined function call**

Processing Unfinished Rules

Example (Initial Rule)

$$Reverse([X|Xs]) = [Z|Zs]$$

Problem: unbound variables Y, Ys in rhs, i.e., unfinished

Three possible solutions (successor functions):

1. **partition** the example equations and compute initial rules for each subset
2. treat subterms of the rhs containing unbound variables as **new (sub)problems**
3. replace rhs by a **defined function call**

Partitioning Example Equations, Case Distinction

- choose a position u with a variable in the lhs of the unfinished rule and with different constructors in the lhss of the example equations
- take all example equations with the same constructor at u into the same subset

Example

Example equations:

1. $Reverse([\,]) = [\,]$, 2. $Reverse([X]) = [X]$, 3. $Reverse([X, Y]) = [Y, X]$

Unfinished initial rule: $Reverse(Xs) = Zs$

u is root position of the pattern, different constructors $[\,]$, $cons$
Partition: $\{\{1\}\{2, 3\}\}$
New initial rules: $Reverse([\,]) = [\,]$, $Reverse([X|Xs]) = [Z|Zs]$

Partitioning Example Equations, Case Distinction

- choose a position u with a variable in the lhs of the unfinished rule and with different constructors in the lhss of the example equations
- take all example equations with the same constructor at u into the same subset

Example

Example equations:

1. $Reverse([\,]) = [\,]$, 2. $Reverse([X]) = [X]$, 3. $Reverse([X, Y]) = [Y, X]$

Unfinished initial rule: $Reverse(Xs) = Zs$

u is root position of the pattern, different constructors $[\,]$, *cons*
Partition: $\{\{1\}\{2, 3\}\}$
New initial rules: $Reverse([\,]) = [\,]$, $Reverse([X|Xs]) = [Z|Zs]$

Dealing with Unfinished Subterms

- for an unfinished initial rule $F(p) = t$ and corresponding example equations $F(i) = o$:
- replace unfinished subterms $s_1, \ldots, s_k$ in the rhs t by calls to new subprograms $S_1(p), \ldots, S_k(p)$ (finishes the rule)
- induce each S_j from example equations $S_j(i) = o|_u$ with u the position of subterm s_j in t

Dealing with Unfinished Subterms, Example

Example equations:

$$Reverse([X]) = [X], \quad Reverse([X, Y]) = [Y, X]$$

Initial Rule:

$$Reverse([X|Xs]) = [Z|Zs]$$

Finished Rule:

$$Reverse([X|Xs]) = [\, S_1([X|Xs]) \mid S_2([X|Xs])\,]$$

Example equations for S_1, S_2:

$$\begin{array}{llll} S_1([X]) & = X & \qquad S_2([X]) & = [\,] \\ S_1([X, Y]) & = Y & \qquad S_2([X, Y]) & = [X] \end{array}$$

Initial rules for S_1, S_2:

$$S_1([X|Xs]) = Z$$
$$S_2([X|Xs]) = Zs$$

Introduce a Defined Function Call

1. for a fixed defined function F' (to be called) and for each (currently considered) example equation $F(i) = o$ choose a (matched) example equation $F'(i') = o'$ such that $o = o'\tau$
2. replace the rhs t of the unfinished rule $F(p) = t$ by $F'(S(p))$ (finishes it) for a new subprogram S
3. induce S from example equations $S(i) = i'\tau'$

Defined Function Call, Example

Unfinished rule:

$$S_2([X|Xs]) = Zs$$

Example equations:

$$S_2([X]) = [\,], \quad S_2([X, Y]) = [X]$$

Matched *Reverse*-examples:

$$Reverse([\,]) = [\,], \quad Reverse([X]) = [X]$$

Finished equation:

$$S_2([X|Xs]) = Reverse(S_3([X|Xs]))$$

Examples for S_3:

$$S_3([X]) = [\,], \quad S_3([X, Y]) = [X]$$

Initial rule for S_3:

$$S_3([X|Xs]) = Zs$$

Current State of the *Reverse* Example

Example (Current Hypothesis)

$$
\begin{aligned}
Reverse([\,]) &= [\,] \\
Reverse([X|Xs]) &= [\, S_1([X|Xs]) \mid S_2([X|Xs]) \,] \\
S_1([X|Xs]) &= Y \\
S_2([X|Xs]) &= Reverse(S_3([X|Xs])) \\
S_3([X|Xs]) &= Y
\end{aligned}
$$

Example (Example Equations for S_1, S_3)

$$
\begin{aligned}
S_1([X]) &= X & \qquad S_3([X]) &= [\,] \\
S_1([X, Y]) &= Y & \qquad S_3([X, Y]) &= [X]
\end{aligned}
$$

The Induced *Reverse* Program

Suppose we would have given *Last* as background knowledge. Then S_1 will match *Last*. S_3 will become the *Init* function.

Example (Induced *Reverse* Program)

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

Outline

1 Inductive (Functional) Programming
- Introduction

2 Our Approach
- Basic Concepts
- The IGOR2 Algorithm
- Empirical Results and Further Research

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 in about one minute

Future Research

- relaxing the requirement of complete example sets
- heuristics, algorithm schemas, more powerfull kinds of specifications in order to reduce search costs
- higher-order functions

Summary

- "pure" analytical approaches too restrictive, 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: IGOR2
- IGOR2 is more powerfull than existing analytical approaches to inductive programming and on some tested problems more time efficient than existing generate-and-test based systems