SC7: Inductive Programming and Knowledge-level Learning

Ute Schmid

Cognitive Systems
Fakultät Wirtschaftsinformatik und Angewandte Informatik
Otto-Friedrich Universität Bamberg
1 Preliminaries

2 What is Program Synthesis?

3 Approaches to Program Synthesis

4 Example

5 Basic Concepts of IP

6 Analytical vs. Generate-and-Test

7 Example: Analytical IP (Thesys)
Knowledge Level Learning

- opposed to low-level (statistical) learning
- learning as generalization of symbol structures (rules) from experience
- “white-box” learning: learned hypotheses are verbalizable, can be inspected, communicated

Examples:

- **Classification/Concepts:**
  - IF odor=almond THEN poisonous
  - IF cap-shape=conical & gill-color=grey THEN poisonous

- **Recursive Concepts:**
  - ancestor(X,Y) :- parent(X,Y).
  - ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

- **Simple action rules:**
  - IF distance-to-object < threshold THEN STOP

- **Recursive action rules:**
  - e.g., Tower of Hanoi (to be seen later)
# Approaches to Symbolical Learning

## Machine Learning Approaches

- Grammar Inference
- Decision Tree Algorithms
- Inductive Logic Programming
- Evolutionary Programming
- Inductive (functional) Programming

## Inductive Programming

- Very special branch of machine learning
- Learning programs from *incomplete* specifications, typically I/O examples or constraints
Overview

- **Lecture 1: Introduction to IP**
  - Background
  - Basic Concepts
  - Summers’ THESYS system

- **Lecture 2: Approaches to IP**
  - Evolutionary Programming
  - Inductive Logic Programming

- **Lecture 3: Analytical Inductive Programming**
  - The IGOR system
  - Applications to Cognitive Tasks
References

Websites:

http://www.inductive-programming.org/
http://www.cogsys.wiai.uni-bamberg.de/aaip/

Books/Handbook Contributions/Special Issues:


References

Articles:


Bi-annual Workshops

Approaches and Applications of Inductive Programming

- AAIP 2005: associated with ICML (Bonn)
  invited speakers: S. Muggleton, M. Hutter, F. Wysotzki

- AAIP 2007: associated with ECML (Warsaw)
  invited speakers: R. Olsson, L. Hamel

- AAIP 2009: associated with ICFP (Edinburgh)
  Proceedings: Springer Lecture Notes in Computer Science 5812

- AAIP 2011: associated with PPDP 2011 and LOPSTR 2011 (Odense)
  invited speaker: Ras Bodik

- AAIP 2013: Dagstuhl Seminare in December

Community Page

www.inductive-programming.org
Program Synthesis

Automagic Programming

- Let the computer program itself
- Automatic code generation from (non-executable) specifications very high level programming
- Not intended for software development in the large but for semi-automated synthesis of functions, modules, program parts
Approaches to Program Synthesis

Deductive and transformational program synthesis

- Complete formal specifications (vertical program synthesis)
- e.g. KIDS (D. Smith)
- High level of formal education is needed to write specifications
- Tedious work to provide the necessary axioms (domain, types, ...)
- Very complex search spaces

\[ \forall x \exists y \ p(x) \rightarrow q(x, y) \]

\[ \forall x \ p(x) \rightarrow q(x, f(x)) \]

Example

\[ \text{last}(l) \iff \text{find } z \text{ such that for some } y, \ l = y \circ [z] \text{ where } \text{islist}(l) \text{ and } l \neq [ ] \] (Manna & Waldinger)
Inductive program synthesis

- Very special branch of machine learning
- Learning programs from *incomplete* specifications, typically I/O examples or constraints
- Inductive programming (IP) for short

(Flener & Schmid, AI Review, 29(1), 2009; Encyclopedia of Machine Learning, 2010; Schmid, Kitzelmann & Plasmeijr, AAIP 2009)
Artificial Intelligence
Modeling human programming
knowledge, skills, strategies

Software Engineering
Automated code generation
example-driven programming

Inductive Programming
Programs from Examples

Machine Learning
Inductive Inference of
Functional/Declarative
Programming
incomplete specifications

Code generation from
Inductive Programming

Functional/Declarative Programming
Inductive Programming Example

Learning last

I/O Examples

last [a] = a
last [a,b] = b
last [a,b,c] = c
last [a,b,c,d] = d

Generalized Program

last [x] = x
last (x:xs) = last xs

Some Syntax

-- sugared
[1,2,3,4]

-- normal infix
(1:2:3:4:[])

-- normal prefix
((::) 1
  ((::) 2
    ((::) 3
      ((::) 4
        []))))
Inductive Programming – Basics

IP is search in a class of programs (hypothesis space)

Program Class characterized by:

Syntactic building blocks:
- Primitives, usually data constructors
- Background Knowledge, additional, problem specific, user defined functions
- Additional Functions, automatically generated

Restriction Bias
syntactic restrictions of programs in a given language

Result influenced by:

Preference Bias
choice between syntactically different hypotheses
Typical for declarative languages (Lisp, Prolog, ML, Haskell)

Goal: finding a program which covers all input/output examples correctly (no PAC learning) and (recursively) generalizes over them

Two main approaches:

▶ Analytical, data-driven:
  detect regularities in the I/O examples (or traces generated from them) and generalize over them (folding)

▶ Generate-and-test:
  generate syntactically correct (partial) programs, examples only used for testing
Generate-and-test approaches

- ILP (90ies): FFOIL (Quinlan) (sequential covering)
- evolutionary: ADATE (Olsson)
- enumerative: MAGICHASKEWER (Katayama)
- also in functional/generic programming context: automated generation of instances for data types in the model-based test tool G∀st (Koopmann & Plasmeijer)
Inductive Programming – Approaches

Analytical Approaches

- Classical work (70ies–80ies):
  \textsc{Thesys} (Summers), Biermann, Kodratoff
  learn linear recursive Lisp programs from traces

- ILP (90ies):
  Golem, Progol (Muggleton), Dialogs (Flener)
  inverse resolution, $\Theta$-subsumption, schema-guided

- \textsc{Igor1} (Schmid, Kitzelmann; extension of \textsc{Thesys})
  \textsc{Igor2} (Kitzelmann, Hofmann, Schmid)
Summers’ Thesys

(Summers (1977), A methodology for LISP program construction from examples, Journal ACM)

Two Step Approach

- Step 1: Generate traces from I/O examples
- Step 2: Fold traces into recursion

Generate Traces

- Restriction of input and output to nested lists
- Background Knowledge:
  - Partial order over lists
  - Primitives: atom, cons, car, cdr, nil
- Rewriting algorithm with unique result for each I/O pair: characterize I by its structure (lhs), represent O by expression over I (rhs)

→ restriction of synthesis to structural problems over lists (abstraction over elements of a list) not possible to induce member or sort
Example: Rewrite to Traces

I/O Examples

- nil → nil
- (A) → ((A))
- (A B) → ((A) (B))
- (A B C) → ((A) (B) (C))

Traces

\[ F_L(x) \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \text{atom}((\text{cdr}(x))) \rightarrow \text{cons}(x, \text{nil}), \text{atom}((\text{cddr}(x))) \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil}))), T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil}))))) \]
Example: Deriving Fragments

Unique Expressions for Fragment \((A \ B)\)

\[(x, (A \ B)),\]
\[(\text{car}[x], A),\]
\[(\text{cdr}[x], (B)),\]
\[(\text{cadr}[x], B),\]
\[(\text{cddr}[x], ( ))\]

Combining Expressions

\[((A) \ (B)) = \text{cons}[(A); ((B))]] = \text{cons}[	ext{cons}[A, ()];\text{cons}[(B), ( )]].\]

Replacing Values by Functions

\[\text{cons}[	ext{cons}[	ext{car}[x]; ( )];\text{cons}[	ext{cdr}[x]; ( )]].\]
Folding of Traces

- Based on a program scheme for linear recursion (restriction bias)
- Synthesis theorem as justification
- Idea: inverse of fixpoint theorem for linear recursion
- Traces are $k$th unfolding of an unknown program following the program scheme
- Identify differences, detect recurrence

$$F(x) \leftarrow (p_1(x) \rightarrow f_1(x), \ldots, p_k(x) \rightarrow f_k(x), T \rightarrow C(F(b(x)), x))$$
Example: Fold Traces

kth unfolding

\[ F_L(x) \leftarrow \]

\[
(\text{atom}(x) \rightarrow \text{nil},
\text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}),
\text{atom}(\text{cddr}(x)) \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil})),
T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil}))))
\]

Differences:

\[
p_2(x) = p_1(\text{cdr}(x))
\]

\[
p_3(x) = p_2(\text{cdr}(x))
\]

\[
p_4(x) = p_3(\text{cdr}(x))
\]

Recurrence Relations:

\[
p_1(x) = \text{atom}(x)
\]

\[
p_{k+1}(x) = p_k(\text{cdr}(x)) \text{ for } k = 1, 2, 3
\]

\[
f_2(x) = \text{cons}(x, f_1(x))
\]

\[
f_3(x) =
\text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_2(\text{cdr}(x))))
\]

\[
f_4(x) =
\text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_3(\text{cdr}(x))))
\]

\[
f_1(x) = \text{nil}
\]

\[
f_2(x) = \text{cons}(x, f_1(x))
\]

\[
f_{k+1}(x) = \text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_k(\text{cdr}(x))))
\]

for \( k = 2, 3 \)
Example: Fold Traces

kth unfolding

\[ F_L(x) \leftarrow \begin{align*} & (\text{atom}(x) \rightarrow \text{nil}, \\
& \quad \text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \\
& \quad \text{atom}(\text{cddr}(x)) \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil})), \\
& \quad T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil})))) \end{align*} \]

Folded Program

\[ \begin{align*} \text{unpack}(x) & \leftarrow \begin{align*} & (\text{atom}(x) \rightarrow \text{nil}, \\
& \quad T \rightarrow \text{u}(x)) \end{align*} \\
\text{u}(x) & \leftarrow \begin{align*} & (\text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \\
& \quad T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{u}(\text{cdr}(x)))) \end{align*} \]
Summers’ Synthesis Theorem

- Based on fixpoint theory of functional program language semantics. (Kleene sequence of function approximations: a partial order can be defined over the approximations, there exists a supremum, i.e. least fixpoint)
- Idea: If we assume that a given trace is the $k$-th unfolding of an unknown linear recursive function, than there must be regular differences which constitute the stepwise unfoldings and in consequence, the trace can be generalized (folded) into a recursive function
Illustration of Kleene Sequence

Defined for no input

\[ U^0 \leftarrow \Omega \]

Defined for empty list

\[ U^1 \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \quad T \rightarrow \Omega) \]

Defined for empty list and lists with one element

\[ U^2 \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \quad \text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \quad T \rightarrow \Omega) \]

\ldots Defined for lists up to \( n \) elements