SC7: Inductive Programming and Knowledge-level Learning

Ute Schmid

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

IK'13

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:

Pierre Flener, 1994, *Logic Program Synthesis from Incomplete Information*, Kluwer.

Alan Biermann, Gerard Guiho, Yves Kodratoff (eds.), 1984, *Automated Program Construction Techniques*, Macmillan.

Ute Schmid, 2003, *Inductive Synthesis of Functional Programs*, Springer LNAI 2654.

Pierre Flener, Ute Schmid, 2010, Inductive Programming. In: Claude Sammut, Geoffrey Webb (eds.), *Encyclopedia of Machine Learning*. Springer.

Allen Cypher (ed.), 1994, *Watch What I Do: Programming by Demonstration*, MIT Press.

Emanuel Kitzelmann, 2010. *A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs*. Dissertationsschrift, Universität Bamberg, Fakultät Wirtschaftsinformatik und Angewandte Informatik.

P. Flener and D. Partridge (guest eds.), 2001, *Special Issue on Inductive Programming*, Automated Software Engineering, 8(2).

References

Articles:

Pierre Flener, Ute Schmid, 2009, An Introduction to Inductive Programming, *Artificial Intelligence Review 29 (1)*, 45-62.

Ute Schmid, Emanuel Kitzelmann, 2011. Inductive Rule Learning on the Knowledge Level. *Cognitive Systems Research 12 (3)*, 237-248.

Emanuel Kitzelmann (2009). Inductive Programming - A Survey of Program Synthesis Techniques. In: Ute Schmid, Emanuel Kitzelmann, Rinus Plasmeijer (eds.): *Proceedings of the ACM SIGPLAN Workshop on Approaches and Applications of Inductive Programming* (AA IP 2009, Edinburgh, Scotland, September 4). Springer LNCS 5812.

IP – Community

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)
  invited speakers: L. Augustsson, N. Mitchell, P. Koopman & R. Plasmeijer
  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

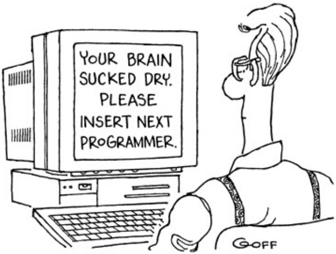
Community Page

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

$$\forall x \exists y \quad p(x) \rightarrow q(x, y)$$

$$\forall x \quad p(x) \rightarrow q(x, f(x))$$

Example

last(l) $\Leftarrow$ *find z such that for some y,* $l = y \circ [z]$
where islist(l) and $l \neq [\,]$ (Manna & Waldinger)

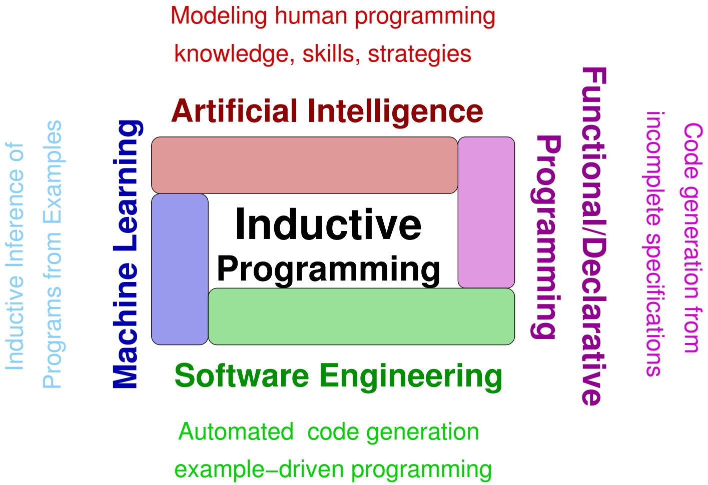
Approaches to Program Synthesis

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)

IP – Contributing Areas



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

Inductive Programming – Approaches

- Typical for declarative languages (LISP, PROLOG, ML, HASKELL)
- Goal: finding a program which covers *all* input/output examples *correctly* (no PAC learning) and (recursivly) 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

Inductive Programming – Approaches

Generate-and-test approaches

- ILP (90ies): FFOIL (Quinlan) (sequential covering)
- evolutionary: ADATE (Olsson)
- enumerative: MAGICHASKELLER (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):
  Thesys (Summers), Biermann, Kodratoff
  learn linear recursive Lisp programs from traces
- ILP (90ies):
  Golem, Progol (Muggleton), Dialogs (Flener)
  inverse resolution, Θ-subsumption, schema-guided
- Igor1 (Schmid, Kitzelmann; extension of Thesys)
  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)

$\hookrightarrow$ 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

$$\text{nil} \rightarrow \text{nil}$$
$$(A) \rightarrow ((A))$$
$$(A\ B) \rightarrow ((A)\ (B))$$
$$(A\ B\ C) \rightarrow ((A)\ (B)\ (C))$$

Traces

$$\begin{aligned} F_L(x) \leftarrow\quad & (\text{atom}(\text{x}) \rightarrow \text{nil}, \\ & \text{atom}(\text{cdr}(\text{x})) \rightarrow \text{cons}(\text{x}, \text{nil}), \\ & \text{atom}(\text{cddr}(\text{x})) \rightarrow \text{cons}(\text{cons}(\text{car}(\text{x}), \text{nil}), \text{cons}(\text{cdr}(\text{x}), \text{nil})), \\ & \text{T} \rightarrow \text{cons}(\text{cons}(\text{car}(\text{x}), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(\text{x}),\text{nil}), \\ & \qquad\qquad \text{cons}(\text{cddr}(\text{x}),\text{nil})))) \end{aligned}$$

Example: Deriving Fragments

Unique Expressions for Fragment (A B)

(x, (A B)),
(car[x], A),
(cdr[x], (B)),
(cadr[x], B),
(cddr[x], ( ))

Combining Expressions

((A) (B)) = cons[(A); ((B))] = cons[cons[A, ()];cons[(B), ( )]].

Replacing Values by Functions

cons[cons(car[x]; ( )];cons[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 recusion
- Traces are kth unfolding of an unknown program following the program scheme
- Identify differences, detect recurrence

$$
\begin{array}{rl}
F(x) \leftarrow & (p_1(x) \rightarrow f_1(x), \\
& \;\ldots, \\
& \;p_k(x) \rightarrow f_k(x), \\
& \;T \rightarrow C(F(b(x)), x))
\end{array}
$$

Example: Fold Traces

kth unfolding

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (\text{atom}(\text{x}) \rightarrow \text{nil}, \\
& \ \text{atom}(\text{cdr}(\text{x})) \rightarrow \text{cons}(\text{x, nil}), \\
& \ \text{atom}(\text{cddr}(\text{x})) \rightarrow \text{cons}(\text{cons}(\text{car}(\text{x}), \text{nil}), \text{cons}(\text{cdr}(\text{x}), \text{nil})), \\
& \ \text{T} \rightarrow \text{cons}(\text{cons}(\text{car}(\text{x}), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(\text{x}), \text{nil}), \\
& \qquad\qquad\quad \text{cons}(\text{cddr}(\text{x}), \text{nil}))))
\end{aligned}
$$

Differences:
$p_2(x) = p_1(\text{cdr}(\text{x}))$
$p_3(x) = p_2(\text{cdr}(\text{x}))$
$p_4(x) = p_3(\text{cdr}(\text{x}))$

$f_2(x) = cons(x, f_1(x))$
$f_3(x) =$
$cons(cons(car(x), nil), f_2(cdr(x)))$
$f_4(x) =$
$cons(cons(car(x), nil), f_3(cdr(x)))$

Recurrence Relations:
$p_1(x) = \text{atom}(\text{x})$
$p_{k+1}(x) = p_k(\text{cdr}(\text{x}))$ for $k = 1, 2, 3$

$f_1(x) = nil$
$f_2(x) = cons(x, f_1(x))$
$f_{k+1}(x) = cons(cons(car(x), nil), f_k(cdr(x)))$
for $k = 2, 3$

Example: Fold Traces

kth unfolding

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (\mathrm{atom}(\mathrm{x}) \rightarrow \mathrm{nil},\\
& \mathrm{atom}(\mathrm{cdr}(\mathrm{x})) \rightarrow \mathrm{cons}(\mathrm{x},\ \mathrm{nil}),\\
& \mathrm{atom}(\mathrm{cddr}(\mathrm{x})) \rightarrow \mathrm{cons}(\mathrm{cons}(\mathrm{car}(\mathrm{x}),\ \mathrm{nil}),\ \mathrm{cons}(\mathrm{cdr}(\mathrm{x}),\ \mathrm{nil})),\\
& \mathrm{T} \rightarrow \mathrm{cons}(\mathrm{cons}(\mathrm{car}(\mathrm{x}),\ \mathrm{nil}),\ \mathrm{cons}(\mathrm{cons}(\mathrm{cadr}(\mathrm{x}),\mathrm{nil}),\\
& \qquad\qquad \mathrm{cons}(\mathrm{cddr}(\mathrm{x}),\mathrm{nil}))))
\end{aligned}
$$

Folded Program

$$
\begin{aligned}
\mathrm{unpack}(\mathrm{x}) \leftarrow \quad & (\mathrm{atom}(\mathrm{x}) \rightarrow \mathrm{nil},\\
& \mathrm{T} \rightarrow \mathrm{u}(\mathrm{x}))\\
\mathrm{u}(\mathrm{x}) \leftarrow \quad & (\mathrm{atom}(\mathrm{cdr}(\mathrm{x})) \rightarrow \mathrm{cons}(\mathrm{x},\ \mathrm{nil}),\\
& \mathrm{T} \rightarrow \mathrm{cons}(\mathrm{cons}(\mathrm{car}(\mathrm{x}),\ \mathrm{nil}),\ \mathrm{u}(\mathrm{cdr}(\mathrm{x}))))
\end{aligned}
$$

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 $\quad U^0 \leftarrow \Omega$

Defined for empty list

$$\begin{aligned} U^1 \leftarrow \ & (atom(x) \rightarrow nil, \\ & T \rightarrow \Omega) \end{aligned}$$

Defined for empty list and lists with one element

$$\begin{aligned} U^2 \leftarrow \ & (atom(x) \rightarrow nil, \\ & atom(cdr(x)) \rightarrow cons(x, nil), \\ & T \rightarrow \Omega) \end{aligned}$$

... Defined for lists up to n elements