SC7: Inductive Programming and Knowledge-level Learning
Lecture 3

Ute Schmid

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

IK'13

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**

IGOR2 is ...

Inductiv

- Induces programs from I/O examples
- Inspired by Summers' THESYS system
- Successor of IGOR1

Analytical

- data-driven
- finds recursive generalization by analyzing I/O examples
- integrates best first search

Functional

- learns functional programs
- first prototype in MAUDE by Emanuel Kitzelmann
- re-implemented in HASKELL and extended (general *fold*) by Martin Hofmann

Some Properties of IGOR2

Hypotheses

- **Termination** of induced programs by construction
- Induced programs are extensionally correct wrt I/O examples
- Arbitrary **user defined data-types**
- **Background knowledge** can (but must not) be used
- **Necessary function invention**
- **Complex call relations** (tree, nested, mutual recursion)
- I/Os with **variables**
- **Restriction bias**: Sub-set of (recursive) functional programs with exclusive patterns, outmost function call is not recursive

Some Properties of IGOR2

Induction Algorithm

- Preference bias: few case distinctions, most specific patterns, few recursive calls
- Needs the *first k* I/O examples wrt input data type
- Enough examples to detect regularities (typically 4 examples are enough for linear list problems)
- Termination guaranteed (worst case: hypothesis is identical to examples)

(Kitzelmann & Schmid, JMLR, 7, 2006; Kitzelmann, LOPSTR, 2008; Kitzelmann doctoral thesis 2010)

Extended Example

reverse

I/O Example

```
reverse  [] =  []        reverse [a,b]   = [b,a]
reverse [a] = [a]        reverse [a,b,c] = [c,b,a]
```

Generalized Program

```
reverse   []   = []
reverse (x:xs) = last (x:xs) : reverse(init (x:xs))
```

Automatically induced functions (*renamed* from $f1$, $f2$)

```
last [x]    = x                    init [a]    = []
last (x:xs) = last xs              init (x:xs) = x:(init xs)
```

Input

Datatype Definitions

```
data [a] = [] | a:[a]
```

Target Function

```
reverse :: [a]-> [a]
reverse       [] =       []
reverse      [a] =      [a]
reverse    [a,b] =    [b,a]
reverse [a,b,c] = [c,b,a]
```

Background Knowledge

```
snoc :: [a] -> a -> [a]
snoc     [] x =      [x]
snoc    [x] y =    [x,y]
snoc [x,y] z = [x,y,z]
```

- Input must be the first k I/O examples (wrt to input data type)
- Background knowledge is *optional*

Output

Set of (recursive) equations which cover the examples

`reverse` Solution

```
reverse    []    = []
reverse (x:xs) = snoc (reverse xs) x
```

Restriction Bias

- Subset of HASKELL
- Case distinction by *pattern matching*
- Syntactical restriction: patterns are not allowed to unify

Preference Bias

- Minimal number of case distinctions

Basic Idea

- Search a rule which explains/covers a (sub-) set of examples
- Initial hypothesis is a single rule which is the least general generalization (anti-unification) over all examples

Example Equations

```
reverse [a]   = [a]
reverse [a,b] = [b,a]
```

Initial Hypothesis

```
reverse (x:xs) = (y:ys)
```

Hypothesis contains *unbound* variables in the body!

Basic Idea cont.

Initiale Hypothesis

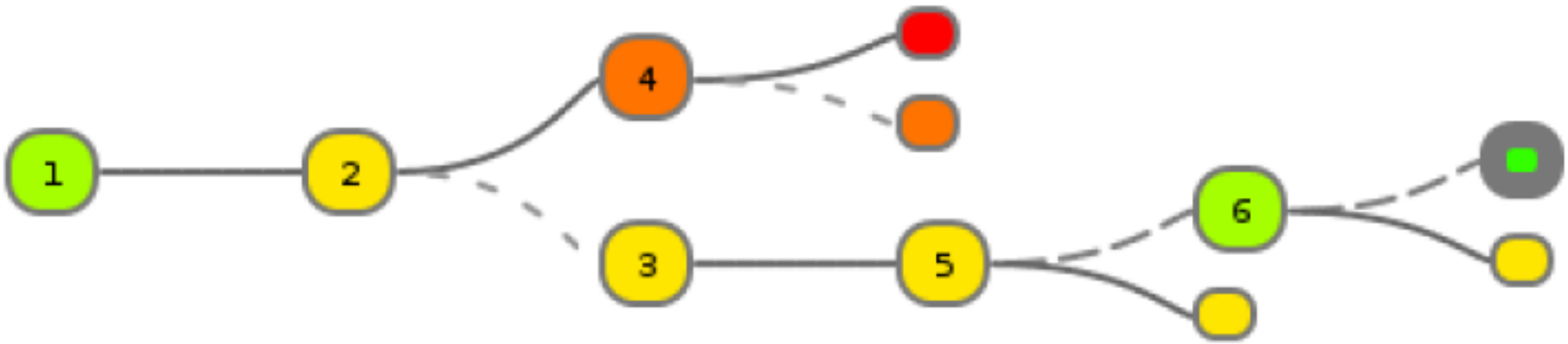
```
reverse (x:xs) = (y:ys)
```

Unbound variables are cue for induction.

Three Induction Operators (to apply simultaneously)

1. **Partitioning** of examples
   $\rightsquigarrow$ *Sets* of equations divided by case distinction
2. Replace righthand side by **program call** (recursive or background)
3. Replace sub-terms with unbound variables by to be induced **sub-functions**

Basic Idea cont.



- In each interation *expand* the *best* hypothesis (due to preference bias)
- Each hypothesis has typically more than one successor

Partitioning, Case Distinction

- Anti-unified terms differ at least at one position wrt constructor
- Partition examples in subsets wrt constructors

Beispiele

```
(1) reverse   []     =   []
(2) reverse (a:[])   = (a:[])
(3) reverse (a:b:[]) = (b:a:[])
```

Anti-unified Term

```
reverse x = y
```

At root positions are constructors `[]` und `(:)`

$\{1\}$

```
reverse []  = []
```

$\{2, 3\}$

```
reverse (x:xs)  = (y:ys)
```

Program Call

```
reverse [a,b] = [b,a]        snoc [x] y = [x,y]
```

$$\Downarrow \{x \leftarrow b, y \leftarrow a\} \Downarrow$$

```
reverse [a,b] = snoc ? ?
```

- If an output corresponds to the output of another function f, the output can be replaced by a call of f
- Constructing the arguments of the function call is a new induction problem
- I/O examples are abduced:
  - Identical inputs
  - Outputs are substituted inputs for the matching output

Program Call – Example

Example Equation:

```
reverse [a,b] = b:[a]
```

Background Knowledge:

```
snoc [x] y = x:[y]
```

(b:[a]) matches (x:[y]) with substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

replace righthand side of reverse

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 calculates 1. argument

fun_2 calculates 2. argument

abduced examples

rhs of reverse and subst. 1./2. argument of snoc

```
fun1  [a,b] = [b]
```

```
fun2  [a,b] = a
```

Sub-Functions

Example equations:

```
reverse [a]   = [a]
reverse [a,b] = [b,a]
```

Initial Hypothesis:

```
reverse (x:xs) = (y:ys)
```

- Each sub-term of the rhs with an unbaound variable is replaced by a call of a (to be induced) sub-function
- I/Os of the sub-functions are abduced
  - Inputs remain as is
  - Outputs are replaced by corresponding sub-terms

Sub-Functions – Examples

Example Equations

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initial hypothesis:

```
reverse (x:xs) = (y : ys)
```

keep constructions and replace variables on rhs

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduced I/Os of sub-functions

```
fun1 [a]   = a          fun2 [a]   =  []
fun1 [a,b] = b          fun2 [a,b] = [a]
```

Some Empirical Results (Hofmann et al. AGI'09)

	isort	*reverse*	*weave*	*shiftr*	*mult/add*	*allodds*
ADATE	70.0	78.0	80.0	18.81	—	214.87
FLIP	×	—	$134.24^{\perp}$	$448.55^{\perp}$	×	×
FFOIL	×	—	$0.4^{\perp}$	$< 0.1^{\perp}$	$8.1^{\perp}$	$0.1^{\perp}$
GOLEM	0.714	—	$0.66^{\perp}$	0.298	—	$0.016^{\perp}$
IGOR II	0.105	0.103	0.200	0.127	⊙	⊙
MAGH.	0.01	0.08	⊙	157.32	—	×

	lasts	*last*	*member*	*odd/even*	*multlast*
ADATE	822.0	0.2	2.0	—	4.3
FLIP	×	0.020	17.868	0.130	$448.90^{\perp}$
FFOIL	$0.7^{\perp}$	0.1	$0.1^{\perp}$	$< 0.1^{\perp}$	< 0.1
GOLEM	1.062	< 0.001	0.033	—	< 0.001
IGOR II	5.695	0.007	0.152	0.019	0.023
MAGH.	19.43	0.01	⊙	—	0.30

— not tested × stack overflow ⊙ timeout $\perp$ wrong

all runtimes in seconds

Evaluation

IGOR2 ...

- is highly efficient and has a larger scope than other analytical systems
- is the only IP system which incorporates learning mutual recursion
- incorporates necessary function invention
- exists in a yet more general variant based on identification of characteristics of higher-order functions (general *fold*) in the examples (doctoral thesis of Martin Hofmann, 2010)

Knowledge Level Learning

- IP approaches learn sets of symbolic rules from experience
- In cognitive architectures, learning is often only addressed on the 'sub-symbolic' level
  - strength values of production rules in ACT-R
  - reinforcement learning in SOAR
  - Bayesian cognitive modeling
- *Where do the rules come from?*

Inductive Inference

- The human ability to master complex demands is to a large extend based on the ability to exploit previous experiences:
  - predict characteristics or reactions of (natural or man-made, inanimate or animate) objects
  - reason about possible outcomes of actions
  - apply previously successful routines and strategies to new tasks and problems
- Often experience is a stream of only positive examples (unsupervised generalization)
- Core process to expand knowledge: Inductive Inference
  that is: construct hypotheses, in such a way that we can transfer knowledge from previous experience to new situations
- e.g. philosophy (Goodman, 1965), AI (machine learning), psychology (Klahr & Wallace, 1976; Tenenbaum, Griffiths, & Kemp, 2006)

  Holland, Holyoak, Nisbett & Thagard (1986). Induction – Processes of inference, learning, and discovery. Cambridge, MA: MIT Press.

Productive Rule Sets

- One specific aspect of inductive inference is the acquisition of **productive rules** from experience
- Chomsky (1965): a system of rules is productive, when it can be applied in situations of various complexity
  - grammar rules (the somewhat outfashioned idea of an LAD)
  - recursive concepts
    odd number, *prime number*, *sorted list*, *ancestor*
  - generation and application of regular action sequences
    blocksworld, *Tower of Hanoi*
- Typically: experience with a small set of examples and induction of an infinite regularity which allows application to new instances of arbitrary complexity
- **Competence** (underlying knowledge) vs. performance (open to unsystematic variations and errors)
- **Verbalizable** knowledge: a recursive concept or a problem solving strategy can be explained to another person

Analytical Inductive Programming

- Analytical inductive programming: Approach to learn recursive programs from small sets of positive input/output examples
- Provides clearly defined restriction and preference biases
- Addresses high-level, symbolic learning
- Can be naturally integrated into rule-based systems
- Captures the ability to generalize by detection of regularities in observations

Proposition

- Analytical inductive programming as a model for learning complex strategies from problem solving experience.
- IGOR (or some other analytical IP approach) as a 'Rule Acquistion Device'

Learning from Problem Solving Experience

Applying IP to Learning from Problem Solving

- Idea: Learn from a problem with small complexity and generalize a recursive rule set which can generate action sequences for problems in the same domain with arbitrary complexity
- e.g., generate a plan for Tower of Hanoi with three discs and generalize to n discs
- IGORII as one (not the only) possible approach
  however, data-driven approach seems to be more plausible than generate-and-test as a cognitive model

(Schmid & Wysotzki, ECML'98; Schmid, Hofmann, Kitzelmann, AGI'2009; Schmid & Wysotzki, AIPS 2000; Schmid, LNAI 2654; Schmid & Kitzelmann CSR, 2011)

Learning Tower of Hanoi

Input to IGOR2

```
eq Hanoi(0, Src, Aux, Dst, S) =
  move(0, Src, Dst, S) .
eq Hanoi(s 0, Src, Aux, Dst, S) =
  move(0, Aux, Dst,
    move(s 0, Src, Dst,
      move(0, Src, Aux, S))) .
eq Hanoi(s s 0, Src, Aux, Dst, S) =
  move(0, Src, Dst,
    move(s 0, Aux, Dst,
      move(0, Aux, Src,
        move(s s 0, Src, Dst,
          move(0, Dst, Aux,
            move(s 0, Src, Aux,
              move(0, Src, Dst, S)))))))) .
```

Induced Tower of Hanoi Rules (3 examples, 0.076 sec)

```
Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S)
Hanoi(s D, Src, Aux, Dst, S) =
  Hanoi(D, Aux, Src, Dst,
    move(s D, Src, Dst,
      Hanoi(D, Src, Dst, Aux, S)))
```

Learning Productive Rule Sets with IGORII

Current State

- Example equations are currently provided by hand
- To do: Combine with a planning algorithm to obtain traces and rewrite suitable for IGORII (a first approach: Schmid, 2003)
- To do: retrieval of a suitable recursive rule set for a planning problem (most simple: by name; a bit more sophisticated: by goals)

Relation to: Planning and Learning

- Plan for a small problem, generalize, no need to apply search for further (more complex) problems in this domain

Learning Productive Rule Sets with IgorII

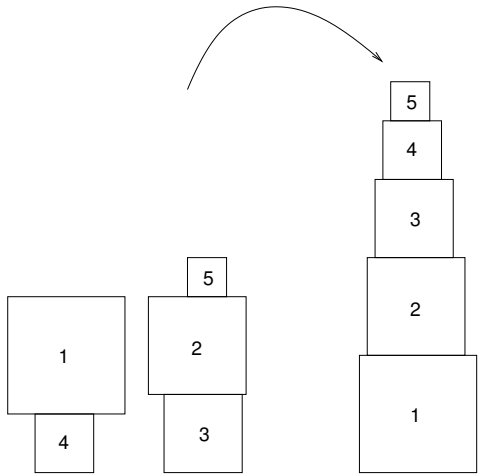
Cognitive Interpretation

- Going beyond learning as chunking or modifying strength values of rules (as in cognitive archictectures)
- Adresses learning of new problem solving strategies
- Acquired recursive rule sets are problem solving schemes, represent verbalizable knowledge, can be integrated in rule-based systems
- IgorII as a "watch dog" in working memory: if content can be generalized, it is done
  corresponds to experiencing sudden insight ("Aha" experience)

Learning to Build a Tower

The Tower Example

- Even small children learn very fast how to stack blocks in a given sequence
- No "stupid" strategies such as first put all blocks on the table and then stack them in the desired order but optimal strategy
- IGOR2 learns `Tower` from 9 examples of towers with up to four blocks in 1.2 sec



One of the 9 Examples

```
eq Tower(s s table,
    ((s s s s table) (s table) table | ,
      (s s s table) (s s table) table | ,
        nil)) =
  put(s s table, s table,
    put(s s s table, table,
      put(s s s s table, table,
        ((s s s s table) (s table) table | ,
          (s s s table) (s s table) table | ,
            nil))))
```

- Examples are equations with the given state specified in the head and the optimal action sequence (generated by a planner) as body
- additionally: 10 corresponding examples for `Clear` and `IsTower` predicate as background knowledge

Generalized Tower Rule Set

```
Tower(O, S) = S if IsTower(O, S)
Tower(O, S) =
  put(O, Sub1(O, S),
    Clear(O, Clear(Sub1(O, S),
      Tower(Sub1(O, S), S))))   if not(IsTower(O, S))
Sub1(s(O), S) = O .
```

Put the desired block x on the one which has to be below y in a situation where both blocks are clear and the blocks up to the block y are already a tower.

Further Examples

Problem Solving

- `Clearblock` (4 examples,0.036 sec)
- `Rocket` (3 examples, 0.012 sec)
- `Tower of Hanoi` (3 examples, 0.076 sec)
- `Car Park` (4 examples, 0.024 sec)

Recursive Concepts

- `isa`
- `odd/even`

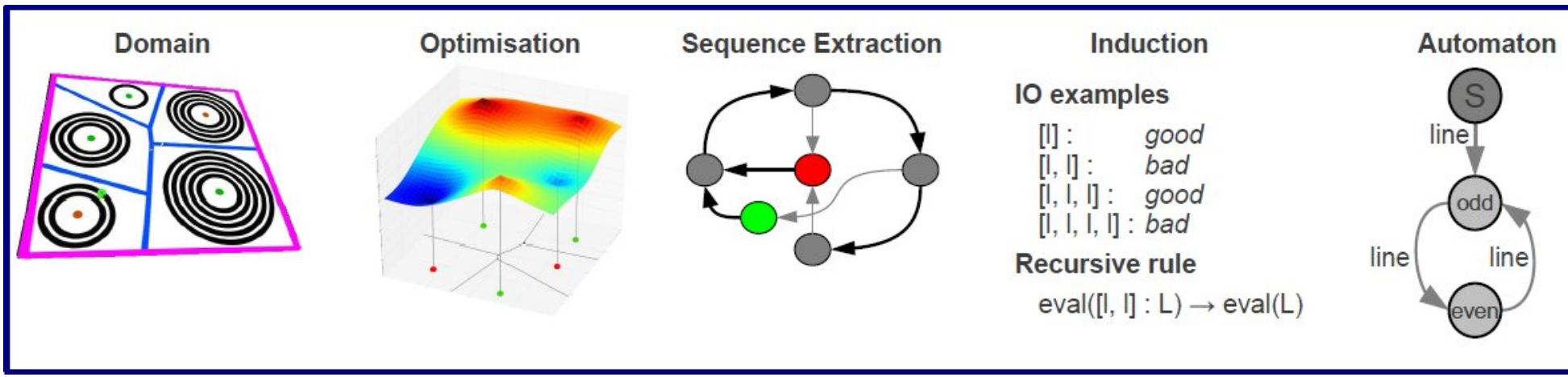
Syntactic Rules

- Phrase structure grammar

Induction of Number Series

Recursive Concepts

- Ongoing work: Combining IGORII with "embedded cognition" (agent learns odd/even in a disc world) (Wernsdorfer, Raab, Schmid & Kitzelmann, 2011)



Learning a Phrase-Structure Grammar

Learning rules for natural language processing: e.g. a phrase structure grammar

1: *The dog chased the cat.*

2: *The girl thought the dog chased the cat.*

3: *The butler said the girl thought the dog chased the cat.*

4: *The gardener claimed the butler said the girl thought the dog chased the cat.*

$$\texttt{S} \rightarrow \texttt{NP VP}$$
$$\texttt{NP} \rightarrow \texttt{d n}$$
$$\texttt{VP} \rightarrow \texttt{v NP} \mid \texttt{v S}$$

Induction of Number Series

- Finding the next number in a sequence of numbers is part of many I.Q. tests
- Ability to detect a regularity and apply it
- Current systems are mostly specifically designed for this problem
- IGOR's induction mechanism works also for induction over number series

Example

Different Representations for PlusTwo

(1) Input List – Output Successor Value

```
eq Plustwo((s 0) nil) = s^3 0
eq Plustwo((s^3 0) (s 0) nil) = s^5 0
eq Plustwo((s^5 0) (s^3 0) (s 0) nil) = s^7 0
```

(2) Input Position – Output List

```
eq Plustwo(s 0) = (s 0) nil
eq Plustwo(s^2 0) = (s^3 0)(s 0) nil
eq Plustwo(s^3 0) = (s^5 0)(s^3 0)(s 0) nil
eq Plustwo(s^4 0) =
            (s^7 0)(s^5 0)(s^3 0)(s 0) nil
```

(3) Input Position – Output Value

```
eq Plustwo(s 0) = s 0
eq Plustwo(s s 0) = s s s 0
eq Plustwo(s s s 0) = s s s s s 0
eq Plustwo(s s s s 0) = s s s s s s s 0
```

Example Series

- $(n-i)+c$ with $i = 1..3, c = 1..12$: 15,17,17,27,29,29,39
- $(n-i)*c$ with $i = 1..3, c = 1..3$: 5,15,45,135,405
- $(n-1)*f$ with $f = n-1, n-2, n$: 2,4,12,48,240
- $(n-i)*c_1+c_2$ with $i = 1..3, c_1 = 1..3,\ c_2 = 1..7$: 5,7,8,22,28,31,73

Burghardt Examples (E-Generalization)

0,1,4,9	$v_p * v_p$
0,2,4,6	$s(s(v_p))$
0,2,4,6	$v_p + v_p$
1,1,2,3,5	$v_1 + v_2$
0,1,2,1,4,1	$if\ (ev(v_p); v_p; 1)$ NOT
0,0,1,1,0,0,1,1	$ev(v_2)$
0,0,1,0,0,1	$ev(v_1 + v_2)$
0,1,3,7	$s(v_1 + v_1)$
1,2,2,3,3,3,4,4,4,4	NOT

Example Series

Ragni & Klein Examples (ANN)

By ANN and IGOR: 7,10,9,12,11 $f(n-1)+3, f(n-1)-1$

By IGOR but not by ANN: 3,7,15,31,63 $2 * f(n-1)+1$

By ANN but not by IGOR: 6,9,18,21,42 $f(n-1)+3, f(n-1) * 2$

Not by ANN and not by IGOR: 2,5,9,19,37 $f(n-1) * 2+1, f(n-1) * 2-1$

Wrapping Up

- Inductive programming addresses the problem of generating (declarative) programs from incomplete specifications
- Analytical IP is highly efficient because search is guided by the given examples
- Analytical IP can be applied to learn generalized rule sets for planning domains, making search for a plan obsolete
- Analytical IP can be seen as one proposal for a general cognitive rule acquisition device
- Learning productive rules goes beyond the types of learning typically addressed in cognitive systems