Inductive Functional Programming

Ute Schmid

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

(Part of the IGOR slides from Martin Hofmann)

ExCape 4/13/15
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

\textit{last(l) } \Leftrightarrow \textit{find } z \text{ such that for some } y, \textit{ l = y } \circ [z] \text{ where } \textit{islist(l)} \text{ and } l \neq [ ] \text{ (Manna & Waldinger)}
Approaches to Program Synthesis

Inductive program synthesis

- Roots in artificial intelligence (modeling a human programmer)
- Very special branch of machine learning (few examples, not feature vectors but symbolic expressions, hypotheses need to cover all data)
- 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; Gulwani, Hernandez-Orallo, Kitzelmann, Muggleton, Schmid & Zorn, CACM’15)
Overview

1. Introductory Example
2. Basic Concepts
3. Summers’s Thesys System
4. IGOR2
5. Inductive Programming as Knowledge Level Learning
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
Inductive Programming – Approaches

Generate-and-test approaches

- ILP (90ies): FFOIL (Quinlan) (sequential covering)
- evolutionary: ADATE (Olsson)
- enumerative: MAGICHASKELEL (Katayama)
- also in functional/generic programming context: automated generation of instances for data types in the model-based test tool ∀st (Koopmann & Plasmeijer)
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, $\Theta$-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)

\( \rightarrow \) 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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nil</code></td>
<td><code>nil</code></td>
</tr>
<tr>
<td><code>(A)</code></td>
<td><code>((A))</code></td>
</tr>
<tr>
<td><code>(A B)</code></td>
<td><code>((A) (B))</code></td>
</tr>
<tr>
<td><code>(A B C)</code></td>
<td><code>((A) (B) (C))</code></td>
</tr>
</tbody>
</table>

#### 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)),
(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 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**

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

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:

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

- \( f_1(x) = \text{nil} \)
- \( f_2(x) = \text{cons}(x, f_1(x)) \)
- \( f_{k+1}(x) = \text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_k(\text{cdr}(x))) \) for \( k = 2, 3 \)
Example: Fold Traces

kth unfolding

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

Folded Program

\[ \text{unpack}(x) \leftarrow \begin{align*} 
\text{atom}(x) & \rightarrow \text{nil}, \\
T & \rightarrow \text{u}(x) 
\end{align*} \]

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

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

Defined for empty list and lists with one element

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

... Defined for lists up to \( n \) elements
Time Jump

- IP until mid 1980ies: Synthesis of Lisp programs based on a two-step approach with Thesys as the most successful system
- No break-through, research interest diminished

- 1990ies, success of Inductive Logic Programming (ILP), mainly classifier learning, but also learning recursive clauses
- 1990ies, in another community: evolutionary approaches

- since 2000, new and growing interest in IP
  - New techniques, e.g. Muggleton’s Meta-Interpretive Learning for ILP, Kitzelmann’s analytical approach for IFP, Katayama’s higher-order approach
  - Successful realworld applications, e.g., Gulwani’s FlashFill
- Our IGOR approach: since 1998 (ECAI), back to functional programs, relations to human learning
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 \textbf{Igor2}

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

Hypotheses
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

\[
\begin{align*}
\text{reverse } [] &= [] \\
\text{reverse } [a, b] &= [b, a] \\
\text{reverse } [a] &= [a] \\
\text{reverse } [a, b, c] &= [c, b, a]
\end{align*}
\]

Generalized Program

\[
\begin{align*}
\text{reverse } [] &= [] \\
\text{reverse } (x:xs) &= \text{last } (x:xs) : \text{reverse } (\text{init } (x:xs))
\end{align*}
\]

Automatically induced functions (\textit{renamed} from \textit{f1, f2})

\[
\begin{align*}
\text{last } [x] &= x \\
\text{last } (x:xs) &= \text{last } xs \\
\text{init } [a] &= [] \\
\text{init } (x:xs) &= x:(\text{init } xs)
\end{align*}
\]
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**

\[
\text{reverse } \text{[]} = \text{[]}
\]

\[
\text{reverse } (x:x) = \text{snoc } (\text{reverse } x) x
\]

**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!
Initiale Hypothesis

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

Unbound variables are cue for induction.

Three Induction Operators (to apply simultaneously)

1. **Partitioning** of examples
   ⇝ *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 iteration *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

<table>
<thead>
<tr>
<th>Beispiele</th>
<th>Anti-unified Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) reverse [] = []</td>
<td>reverse x = y</td>
</tr>
<tr>
<td>(2) reverse (a:[]) = (a:[])</td>
<td></td>
</tr>
<tr>
<td>(3) reverse (a:b:[]) = (b:a:[])</td>
<td></td>
</tr>
</tbody>
</table>

At root positions are constructors [] und (:)

<table>
<thead>
<tr>
<th>1</th>
<th>2, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse [] = []</td>
<td>reverse (x:xs) = (y:ys)</td>
</tr>
</tbody>
</table>
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 \([a,b]\))

- **fun1** calculates 1. argument
- **fun2** 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

- \( \text{reverse } [a] = (a: []) \)
- \( \text{reverse } [a, b] = (b: [a]) \)

Initial hypothesis:

- \( \text{reverse } (x:xs) = (y : ys) \)

keep constructions and replace variables on rhs

- \( \text{reverse } (x:xs) = \text{fun1} (x:xs) : \text{fun2} (x:xs) \)

abduced I/Os of sub-functions

- \( \text{fun1 } [a] = a \)
- \( \text{fun1 } [a, b] = b \)
- \( \text{fun2 } [a] = [] \)
- \( \text{fun2 } [a, b] = [a] \)
## Some Empirical Results (Hofmann et al. AGI’09)

<table>
<thead>
<tr>
<th></th>
<th>isort</th>
<th>reverse</th>
<th>weave</th>
<th>shiftr</th>
<th>mult/add</th>
<th>allodds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADATE</strong></td>
<td>70.0</td>
<td>78.0</td>
<td>80.0</td>
<td>18.81</td>
<td>—</td>
<td>214.87</td>
</tr>
<tr>
<td><strong>FLIP</strong></td>
<td>×</td>
<td>—</td>
<td>134.24</td>
<td>448.55</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>FFOIL</strong></td>
<td>×</td>
<td>—</td>
<td>0.4</td>
<td>&lt; 0.1</td>
<td>8.1</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>GOLEM</strong></td>
<td>0.714</td>
<td>—</td>
<td>0.66</td>
<td>0.298</td>
<td>—</td>
<td>0.016</td>
</tr>
<tr>
<td><strong>IGOR II</strong></td>
<td>0.105</td>
<td>0.103</td>
<td>0.200</td>
<td>0.127</td>
<td>⊙</td>
<td>⊙</td>
</tr>
<tr>
<td><strong>MAGH.</strong></td>
<td>0.01</td>
<td>0.08</td>
<td>⊙</td>
<td>157.32</td>
<td>—</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>lasts</th>
<th>last</th>
<th>member</th>
<th>odd/even</th>
<th>multlast</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADATE</strong></td>
<td>822.0</td>
<td>0.2</td>
<td>2.0</td>
<td>—</td>
<td>4.3</td>
</tr>
<tr>
<td><strong>FLIP</strong></td>
<td>×</td>
<td>0.020</td>
<td>17.868</td>
<td>0.130</td>
<td>448.90</td>
</tr>
<tr>
<td><strong>FFOIL</strong></td>
<td>0.7</td>
<td>0.1</td>
<td>0.1</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td><strong>GOLEM</strong></td>
<td>1.062</td>
<td>&lt; 0.001</td>
<td>0.033</td>
<td>—</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td><strong>IGOR II</strong></td>
<td>5.695</td>
<td>0.007</td>
<td>0.152</td>
<td>0.019</td>
<td>0.023</td>
</tr>
<tr>
<td><strong>MAGH.</strong></td>
<td>19.43</td>
<td>0.01</td>
<td>⊙</td>
<td>—</td>
<td>0.30</td>
</tr>
</tbody>
</table>

— not tested × stack overflow ⊙ timeout ⊥ wrong

all runtimes in seconds
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)
- Has been used to model human learning on the knowledge level
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

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?
IP approaches learn sets of symbolic rules from experience!
Learning Productive Rules from Experience

- 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
  - Generate a plan for Tower of Hanoi with three discs and generalize to $n$ discs
  - Being told your ancestor relations up to your great-great-great-grandfather and generalize the recursive concept
  - Get exposed to natural language sentences and learn the underlying grammatical rule

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 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.

S → NP VP
NP → d n
VP → v NP | v S
Solving Number Series Problems

Example Series: [1, 3, 5]

\[ eq \text{Plustwo}((s \ 0) \ \text{nil}) = s^3 \ 0 \]
\[ eq \text{Plustwo}((s^3 \ 0) \ (s \ 0) \ \text{nil}) = s^5 \ 0 \]
\[ eq \text{Plustwo}((s^5 \ 0) \ (s^3 \ 0) \ (s \ 0) \ \text{nil}) = s^7 \ 0 \]

Rule:

\[ eq \text{Plustwo} [s[0:MyNat], \text{Nil:MyList}] = s[s[s[0:MyNat]]] \]
\[ eq \text{Plustwo} [s[s[s[X0:MyNat]]],X1:MyList] = s[s[s[s[X0:MyNat]]]] \]

<table>
<thead>
<tr>
<th>Series Type</th>
<th>Values</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>15 15 16 15 15 16 15</td>
<td>( f(n-3) )</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>2 3 8 11 14</td>
<td>( f(n-1) + 3 )</td>
</tr>
<tr>
<td></td>
<td>1 2 3 12 13 14 23</td>
<td>( f(n-3) + 11 )</td>
</tr>
<tr>
<td>Geometric</td>
<td>3 6 12 24</td>
<td>( f(n-1) \times 2 )</td>
</tr>
<tr>
<td></td>
<td>6 7 8 18 21 24 54</td>
<td>( f(n-3) \times 3 )</td>
</tr>
<tr>
<td></td>
<td>5 10 30 120 600</td>
<td>( f(n-1) \times n )</td>
</tr>
<tr>
<td></td>
<td>3,7,15,31,63</td>
<td>( 2 \times f(n-1) + 1 )</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1 2 3 5 8 13 21 34</td>
<td>( f(n-1) + f(n-2) )</td>
</tr>
<tr>
<td></td>
<td>3 4 12 48 576</td>
<td>( f(n-1) \times f(n-2) )</td>
</tr>
</tbody>
</table>
Wrapping Up

- IP research provides intelligent algorithmic approaches to induce programs from examples.
- An early system learning linear recursive Lisp programs was Thesys.
- A current approach for learning functional Maude or Haskell programs is Igor2.
- Learning recursive programs is a very special branch of machine learning: not based on feature vectors but on symbolic expressions, hypotheses must cover all examples, learning from few data (not big data).
- Learning productive rule sets can be applied to domains outside programming such as learning from problem solving traces, learning regularities in number series.
References

Website:

http://www.inductive-programming.org/

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 Seminar 13502, Dec. 8-11 (organized by Emanuel Kitzelmann, Sumit Gulwani, Ute Schmid, with more than 40 participants)

- **AAIP 2015**: Dagstuhl Seminar 15442, Oct. 25-30 (organized by Jose Hernandez-Orallo, Stephen Muggleton, Ute Schmid, Ben Zorn)