Inductive Program Synthesis

Ute Schmid
Joint work with Emanuel Kitzelmann and Martin Hofmann
IGOR2 slides by Martin Hofmann

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

Bonn, 13.12.2010

1. Introduction
   - Analytical vs. Generate-and-Test
   - Example: Generate-and-test (FFOIL)
   - Example: Analytical IP (Thesys)

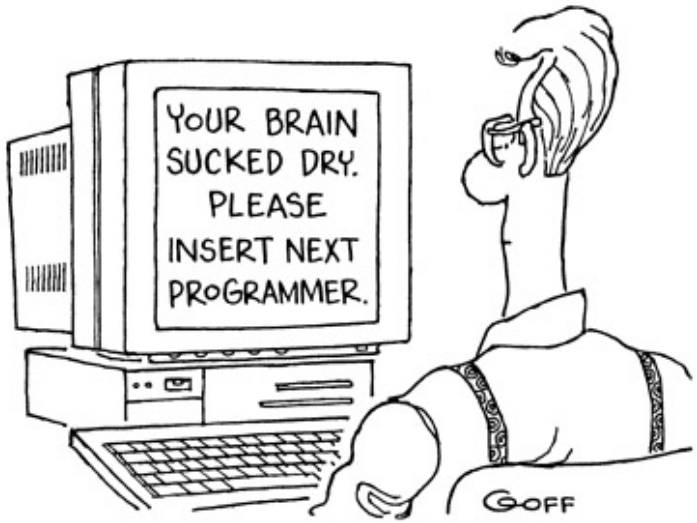
2. IGOR2
   - Basic Idea
   - Operators
   - Evaluation

3. Planning and IP
   - Learning to Build a Tower
   - Igor as General Rule Acquisition Device

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

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

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

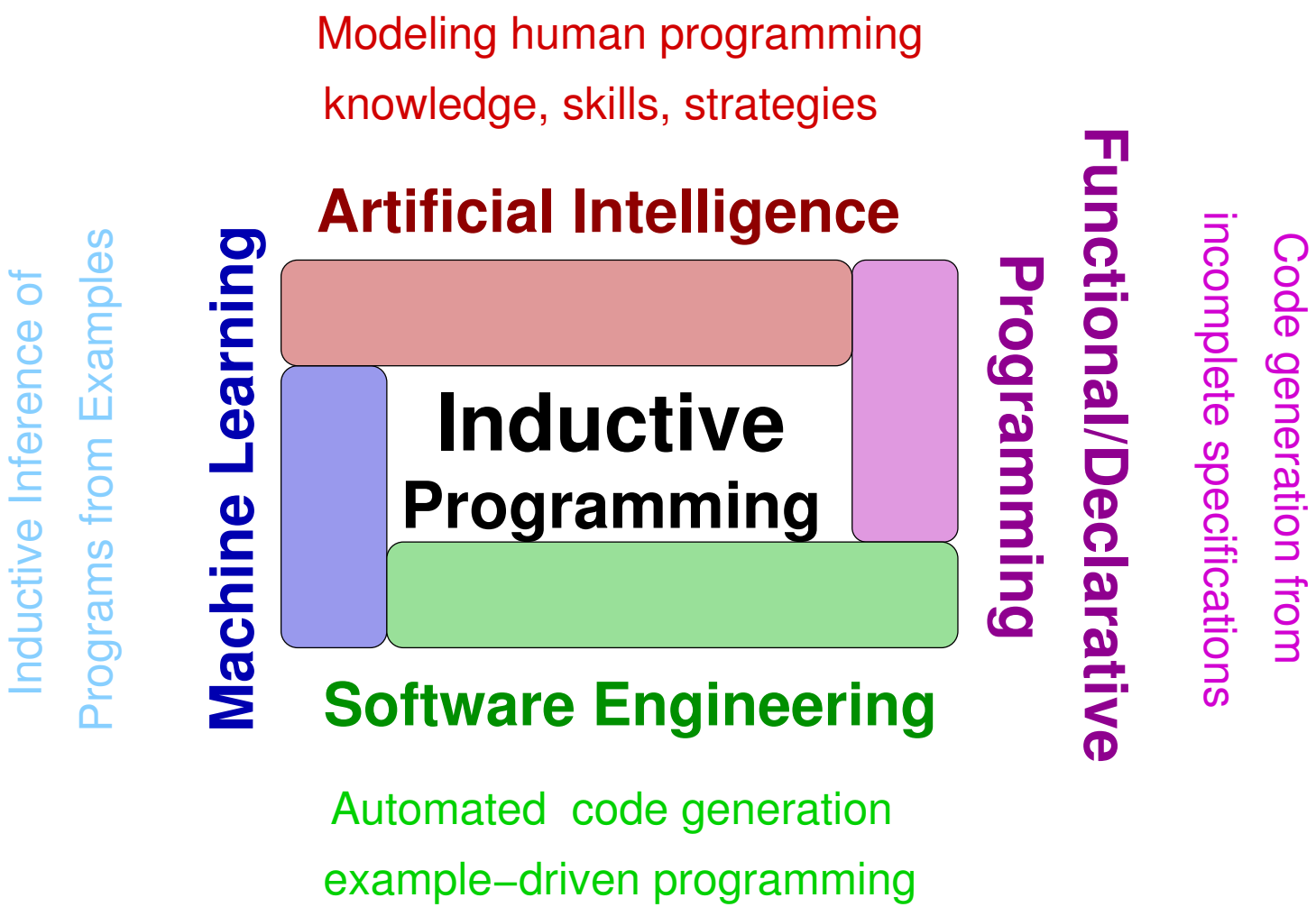
Approaches to Program Synthesis

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, to appear; Schmid, Kitzelmann & Plasmeijr, AAIP 2009)

IP – Contributing Areas



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

Community Page

`www.inductive-programming.org`

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)

FFOIL – Sequential Covering

Initialization:

- DEFINITION := *null program*
- REMAINING := *all tuples belonging to target relation R*

While REMAINING *is not empty*

/* Grow a new clause */

CLAUSE := $R(A, B, \ldots)$:- .

While CLAUSE *has wrong or undefined bindings*

/* Specialize clause */
Find appropriate literal(s) *L*
Add *L* to body of CLAUSE

Remove from REMAINING *tuples in R covered by* CLAUSE

Add CLAUSE *to* DEFINITION

Quinlan (1996), Learning First-Order Definitions for Functions, JAIR

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

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

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

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

Example: Fold Traces

kth unfolding

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

Differences:

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

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

Recurrence Relations:

$p_1(x) = \text{atom(x)}$
$p_{k+1}(x) = p_k(\text{cdr(x)})$ 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

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

Folded Program

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

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

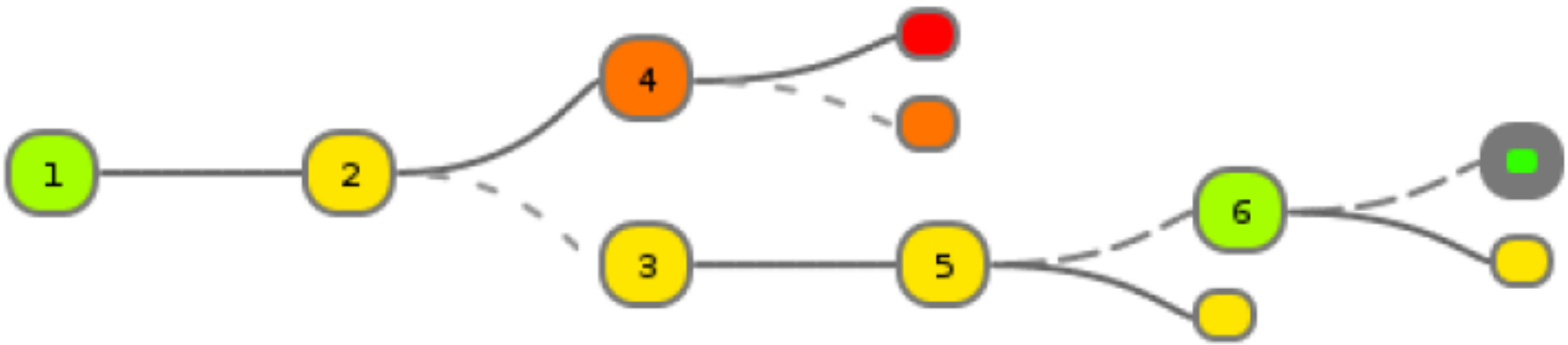
```
reverse (x:xs) = (y:ys)
```

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 ⊥ 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)

Learning and Planning

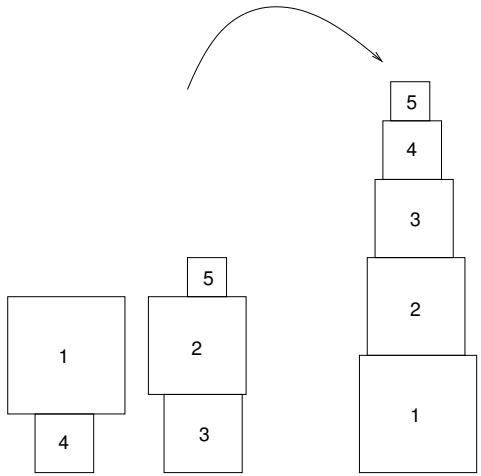
- Domain independent planning has to deal with search spaces of very high complexity
- Idea: Learn from a problem with small complexity and generalize a recursive rule set which can generate action sequences for problems in the same domain with arbitrary complexity
- e.g., generate a plan for Tower of Hanoi with three discs and generalize to n discs

(Schmid, Hofmann, Kitzelmann, AGI'2009; Schmid & Wysotzki, AIPS 2000; Schmid, LNAI 2654; Schmid & Kitzelmann CSR, to appear)

Generalized Rules for Planning

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(0, S) = S if IsTower(0, S)
Tower(0, S) =
  put(0, Sub1(0, S),
    Clear(0, Clear(Sub1(0, S),
      Tower(Sub1(0, S), S))))   if not(IsTower(0, S))
Sub1(s(0), S) = 0 .
```

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.

Generalized Rules for Planning

Further examples;

- `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)

```
eq PutLast(Car1, Curb, Cars, 0, S) = S .
eq PutLast(Car1, Curb, Cars, s 0, S) = move(Car1, Curb, S) .
eq PutLast(Car1, Curb, Car2 Cars, s s 0, S) =
     move(Car1, Curb, move(Car2, Curb, S)) .
eq PutLast(Car1, Curb, Car2 Car3 Cars, s s s 0, S) =
     move(Car1, Curb, move(Car3, Curb, move(Car2, Curb, S))) .

PutLast(Car1, Curb, Cars, 0, S) = S
PutLast(Car1, Curb, Cars, s 0, S) = move(Car1, Curb, S)
PutLast(Car1, Curb, Car2 Cars, s s N, S) =
  PutLast(Car1, Curb, Cars, s N,
    move(Car2, Curb, S))
```

Generalized Rules for Planning

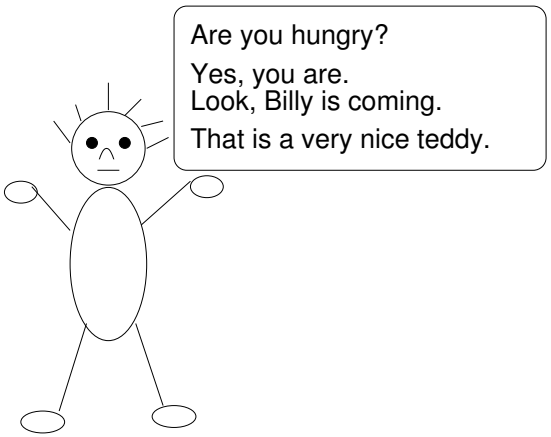
Current State

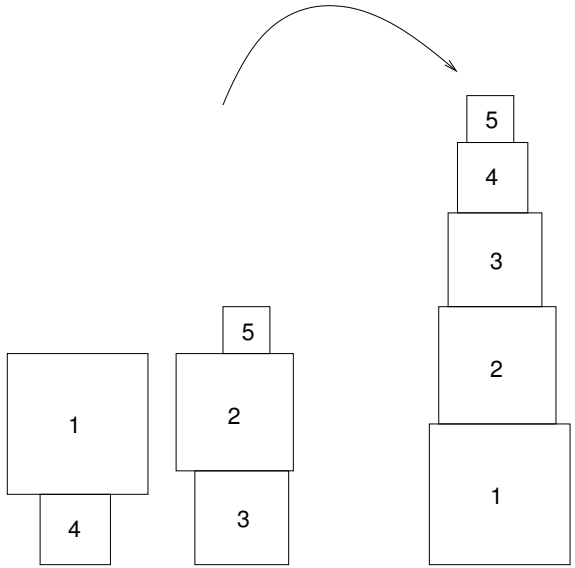
- Example equations are currently hand-crafted
- To do: automatically rewriting plans into traces (first ideas see Schmid, 2003)
- To do: retrieval of suitable recursice rule set for a planning problem

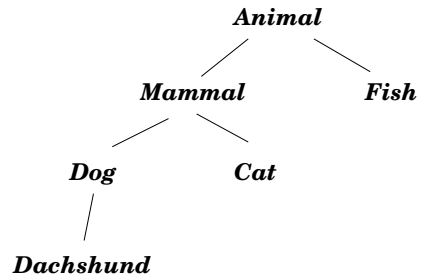
Take a Broader Perspective

Analytical IP provides a mechanism to extract generalized sets of recursive rules from small sets of positive examples of some desired behavior

- Domains where humans are typically exposed to positive examples only:
  - problem solving traces (planning)
  - language
  - semantic relations







From LAD to RAD

- Chomsky's claim of a Language Acquisition Device
- Universal mechanism to extract grammar rules from language experience
- LAD is an inductive learning mechanism for recursive rule sets
- Grammar rules characterize linguistic competence and the systematicity, productivity and compositionality of language

- Analytical IP is an inductive learning mechanism for recursive rule sets with language as a special case
  $\hookrightarrow$ Analytical IP is a possible model for a general cognitive rule acquisition device
- We explore this proposition with IGOR2

IGOR2 (again)

- Efficient induction of recursive rule sets from small sets of positive examples
- Can be applied to learning generalized rules in various cognitive domains, such as problem solving, reasoning, and natural language processing
- Generate-and-test approaches cognitive not plausible
- Cognitive architectures often model learning on the knowledge level simply as chunking of rules or by modifying rule strenghtes (do not explain where the rules come from)
- IGOR2 as a watch dog on working memory: if content contains regularities then generalize
- Acquired recursive rule sets are problem solving schemes, verbalisable knowledge, models sudden insight (Aha!-experience)

Learning from Positive Experience

RAD in Action

- Solve some problems of a domain using a search-based strategy (e.g. planning), observe regularities in the problem solving traces and generalize over them
- For future problems of this domain: application of the learned rules (no need to search anymore = expertise)

↪ Learning is more than chunking or updating strengthes
↪ Learning as acquisition of new problem solving schemes

Learning a Phrase-Structure Grammar

- (Learning rule sets for reasoning, e.g. transitivity of `ancestor`, `isa`)

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.*

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

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
- The scope of analytical IP can be pushed well beyond simple linear recursion
- 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