SC7: Inductive Programming and Knowledge-level Learning
Lecture 2

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

1. Recap: Approaches to IP

2. Evolutionary Programming
   - Genetic Algorithms
   - Genetic Programming Basics
   - Koza's Algorithm
   - Learning to Stack Blocks
   - Adate

3. Inductive Logic Programming
   - Foil
   - FFOIL

Recap: 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

Recap: 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)

Recap: 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)

Background: Genetic Algorithms

- **problem:** search a space of candidate hypotheses to identify the best hypothesis
- the best hypothesis is defined as the one that optimizes a predefined numerical measure, called **fitness**
  - e.g. if the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population
  - in IP: learn a functional program which transfers given inputs into predefined outputs of a training set
- **basic structure:**
  - iteratively updating a pool of hypotheses (**population**)
  - on each iteration
    - hypotheses are evaluated according to the **fitness function**
    - a new population is generated by selecting the most fit individuals
    - some are carried forward, others are used for creating new offspring individuals

Genetic Operators

- generation of successors is determined by a set of operators that recombine and mutate selected members of the current population
- operators correspond to idealized versions of the genetic operations found in biological evolution
- the two most common operators are **crossover** and **mutation**

Hypothesis Space Search

- method is different from typical strategies for search in hypothesis space in machine learning
- neither general-to-specific nor simple-to-complex search is performed
- genetic algorithms can move **very abruptly**, replacing a parent hypothesis by an offspring which is radically different
- so this method is less likely to fall into some local minimum
- practical difficulty: **crowding**
  - some individuals that fit better than others reproduce quickly, so that copies and very similar offspring take over a large fraction of the population

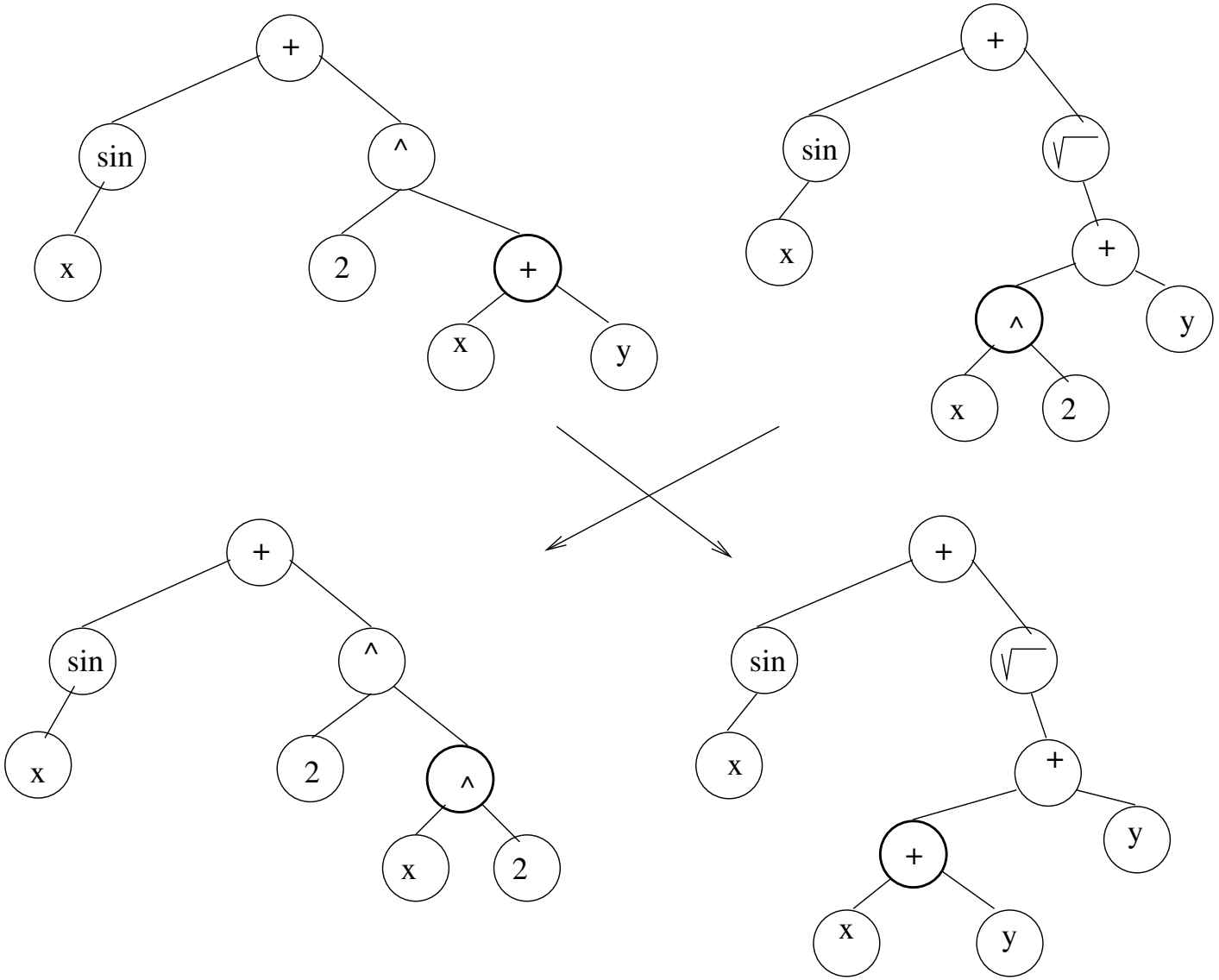
  $\Rightarrow$ reduced diversity of population

  $\Rightarrow$ slower progress of the genetic algorihms

Genetic Programming

- individuals in the evolving population are computer programs (rather than bit strings)
- has shown good results, despite vast H
- **representing programs**
  - typical representations correspond to parse trees
    - each function call is a node
    - arguments are the descendants
  - fitness is determined by executing the programm on the training data
  - crossover are performed by replacing a randomly chosen subtree between parents

Cross-Over



Approaches to Genetic/Evolutionary Programming

- Learning from input/output examples
- Typically: functional programs (term representation)
- Koza (1992): e.g. Stacking blocks, Fibonacci
- Roland Olsson: ADATE, evolutionary computation of ML programs

Koza's Algorithm

1. Generate an initial population of random compositions of the functions and terminals of the problem.
2. Iteratively perform the following sub-steps until the termination criterium has been satisfied:
   1. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   2. Select computer program(s) from the current population chosen with a probability based on fitness.
      Create a new population of computer programs by applying the following two primary operations:
      1. Copy program to the new population (Reproduction).
      2. Create new programs by genetically recombining randomly chosen parts of two existing programs.
3. The best so-far individual (program) is designated as result.

Genetic Operations

Mutation: Delete a subtree of a program and grow a new subtree at its place randomly.
This "asexual" operation is typically performed sparingly, for example with a probability of 1% during each generation.

Crossover: For two programs ("parents"), in each tree a cross-over point is chosen randomly and the subtree rooted at the cross-over point of the first program is deleted and replaced by the subtree rooted at the cross-over point of the second program.
This "sexual recombination" operation is the predominant operation in genetic programming and is performed with a high probability (85% to 90 %).

Genetic Operations

Reproduction:

Copy a single individual into the next generation.
An individuum "survives" with for example 10% probability.

Architecture Alteration:

Change the structure of a program.
There are different structure changing operations which are applied sparingly (1% probability or below):

Introduction of Subroutines:

Create a subroutine from a part of the main program and create a reference between the main program and the new subroutine.

Deletion of Subroutines:

Delete a subroutine; thereby making the hierarchy of subroutines narrower or shallower.

Genetic Operations

Architecture Alteration:

Subroutine Duplication: Duplicate a subroutine, give it a new name and randomly divide the preexisting calls of the subroutine between the old and the new one. (This operation preserves semantics. Later on, each of these subroutines might be changed, for example by mutation.)

Argument Duplication: Duplicate an argument of a subroutine and randomly divide internal references to it. (This operation is also semantics preserving. It enlarges the dimensionality of the subroutine.)

Argument Deletion: Delete an argument; thereby reducing the amount of information available to a subroutine ("generalization").

Genetic Operations

Automatically Defined Iterations/Recursions:

Introduce or delete iterations (ADIs) or recursive calls (ADRs).

Introduction of iterations or recursive calls might result in non-termination. Typically, the number of iterations (or recursions) is restricted for a problem. That is, for each problem, each program has a time-out criterium and is terminated "from outside" after a certain number of iterations.

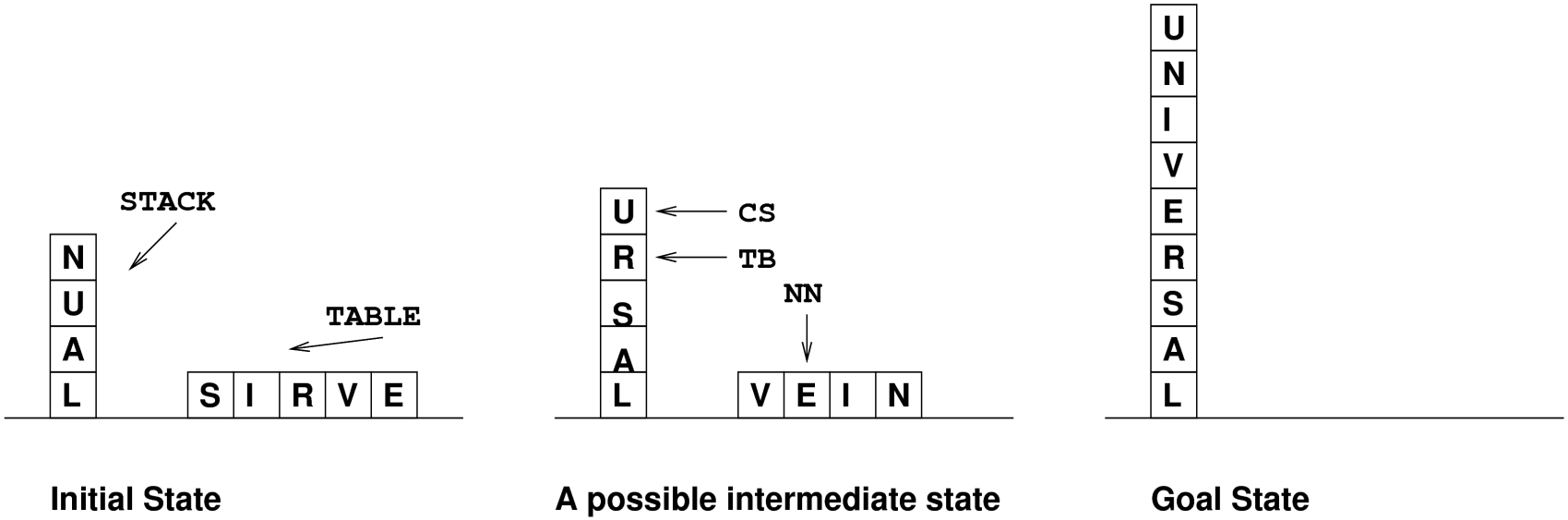
Automatically Defined Stores:

Introduce or delete memory (ADSs).

Fitness

- Quality criteria for programs synthesized by genetic programming:
  - correctness: defined as 100% fitness for the given examples
  - efficiency
  - parsimony
- The two later criteria can be additionally coded in the fitness measure.

Learning to Stack Blocks



Terminals $T = \{CS, TB, NN\}$

CS: A sensor that dynamically specifies the top block of the *Stack*.

TB: A sensor that dynamically specifies the block in the *Stack* which together with all blocks under it are already positioned correctly. ("top correct block")

NN: A sensor that dynamically specifies the block which must be stacked immediately on top of *TB* according to the goal. ("next needed block")

Learning to Stack Blocks

Set of functions $\{MS, MT, NOT, EQ, DU\}$

MS: A move-to-stack operator with arity one which moves a block from the *Table* on top of the *Stack*.

MT: A move-to-table operator with arity one which moves a block from the top of the *Stack* to the *Table*.

NOT: A boolean operator with arity one switching the truth value of its argument.

EQ: A boolean operator with arity two which returns true if its arguments are identical and false otherwise.

DU: A user-defined iterative "do-until" operator with arity two. The expression `DU *Work* *Predicate*` causes `*Work*` to be iteratively executed until `*Predicate*` is satisfied.

Learning to Stack Blocks

- All functions have defined outputs for all conditions: *MS* and *MT* change the *Table* and *Stack* as side-effect. They return *true*, if the operator can be applied successfully and *nil* (*false*) otherwise.
  The return value of *DU* is also a boolean value indicating whether `*Predicate*` is satisfied or whether the *DU* operator timed out.
- Terminals functions carefully crafted. Esp. the pre-defined sensors carry exactly that information which is relevant for solving the problem!
- 166 fitness cases:
  ten cases where zero to all nine blocks in the stack were already ordered correctly; eight cases where there is one out of order block on top of the stack; and a random sampling of 148 additions cases.
- Fitness was measured as the number of correctly handled cases.

Learning to Stack Blocks

- Three variants
  1. first, correct program first moves all blocks on the table and than constructs the correct stack. This program is not very efficient because there are made unnecessary moves from the stack to the table for partially correct stacks. Over all 166 cases, this function generates 2319 moves in contrast to 1642 necessary moves.
  2. In the next trial, efficiency was integrated into the fitness measure and as a consequence, a function calculating only the minimum number of moves emerged. But this function has an outer loop which is not necessary.
  3. By integrating parsimony into the fitness measure, the correct, efficient, and parsimonious function is generated.

Learning to Stack Blocks

Population Size: M = 500
Fitness Cases: 166

Correct Program:
Fitness: *166 - number of correctly handled cases*
Termination: Generation 10

```
(EQ (DU (MT CS) (NOT CS))
    (DU (MS NN) (NOT NN)))
```

Learning to Stack Blocks

- **Correct and Efficient Program:**
  Fitness: $0.75 \cdot C + 0.25 \cdot E$ with
  - $C = ($ *number of correctly handled cases*$/166) \cdot 100$
  - $E = f(n)$ *as function of the total number of moves over all 166 cases:*
  - *with* $f(n) = 100$ *for the analytically obtained minimal number of moves for a correct program (*$min = 1641$*);*
  - $f(n)$ *linearly scaled upwards for zero moves up to 1640 moves with* $f(0) = 0$
  - $f(n)$ *linearly scaled downwards for 1642 moves up to 2319 moves (obtained by the first correct program) and* $f(n) = 0$ *for* $n > 2319$
  - Termination: Generation 11

```
(DU (EQ (DU (MT CS) (EQ CS TB))
        (DU (MS NN) (NOT NN)))
    (NOT NN))
```

Learning to Stack Blocks

Correct, Efficient, and Parsimonious Program:

Fitness: $0.70 \cdot C + 0.20 \cdot E + 0.10 \cdot$ *(number of nodes in program tree)*

Termination: Generation 1

```
(EQ (DU (MT CS) (EQ CS TB))
    (DU (MS NN) (NOT NN)))
```

ADATE

- ongoing research, started in the 90s
- most powerful approach to inductive programming based on evolution
- constructs programs in a subset of the functional language ML, called ADATE-ML
- problem specification:
  - a set of data types and a set of primitive functions
  - a set of sample inputs (input/output pairs; additionally: negative examples can be provided)
  - an evaluation function (different predefined functions available, typicylla including syntactic complexity and time efficiency)
  - an initial declaration of the goal function f
- Start with a single individual – the empty function f.
- Sets of individuals are developed over generations such that fitter individuals have more chances to reproduce

ADATE

- J. R. Olsson, Inductive functional programming using incremental program transformation, *Artificial Intelligence, 74(1)*, 1995, pp. 55–83.
- Succesful synthesis of a sorting program
- Current applications: code optimization
- Combination of ADATE and IGOR: Use fast analytical approach to generate a 'skeleton', evolve from there
  Crossley, Neil, Kitzelmann, Emanuel, Hofmann, Martin, Schmid, Ute, Combining Analytical and Evolutionary Inductive Programming, Proceedings of AGI-09, pp. 19–24. – Winner of the 2009 Kurzweil Best AGI Paper Prize

Inductive Logic Programming

- When progress was slow for Summers' like approaches at the end of the 1980ies, hope was set on inductive logic programming as the "golden method" for inductive programming
- Three directions
  - Generate-and-test, sequential covering: FOIL, FFOIL (Quinlan)
  - Example-driven, based on inverse resolution, Θ-subsumption: GOLEM, PROGOL (Muggleton)
  - Schema-based, interactive: DIALOGS (Flener)

FOIL (Quinlan, 1990)

- Learns PROLOG programs from positive and negative examples for the target function and background knowledge
- Is based on sequential-covering (in contrast to simultaneous covering, ID3)
- Learns separate rules, each covering a sub-set of the positive examples and excluding the negative examples
- For each new rule: the goal predicate is used as head and the body is empty
- The body is extended with literals from the background knowledge (and with the goal literal).
- Which literal is suited best is estimated by an information theoretical measure (FOIL-gain)
- Rule extension is performed with hill-climbing

FOIL

Algorithm

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, Do
  - *NewRule* ← the rule that predicts *Target_predicate* with no precondition
  - *NewRuleNeg* ← *Neg*
  - while *NewRuleNeg*, Do
    - *Candidate_literals* ← generate new literals for *NewRule*, based on *Predicates*
    - *Best_literal* ← $\max_{L \in Candidate\_literals} FoilGain(L, NewRule)$
    - add *Best_literal* to preconditions of *NewRule*
    - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
  - *Learned_rules* ← *Learned_rules* + *NewRule*
  - *Pos* ← *Pos* − { members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

FOIL Hypothesis Space

outer loop (set of rules)

- specific-to-general search
- initially, there are no rules, so that each example will be classified negative (most specific)
- each new rule raises the number of examples classified as positive (more general)
- disjunctive connection of rules

inner loop (preconditions for one rule)

- general-to-specific search
- initially, there are no preconditions, so that each example satisfies the rule (most general)
- each new precondition raises the number of examples classified as negative (more specific)
- conjunctive connection of preconditions

Generating Candidate Specializations

- current rule:

  $P(x_1, x_2, ..., x_k) \leftarrow L_1 ... L_n$ where

  $L_1 ... L_n$ are the preconditions and
  $P(x_1, x_2, ..., x_k)$ is the head of the rule

- FOIL generates candidate specializations by considering new literals L_{n+1} that fit one of the following forms:
  - $Q(v_1, ..., v_r)$ where $Q \in Predicates$ and the v_i are new or already present variables (at least one v_i must already be present)
  - $Equal(x_j, x_k)$ where x_j and x_k are already present in the rule
  - the negation of either of the above forms

Training Data Example

Examples	Background Knowledge
GrandDaughter(Victor, Sharon)	Female(Sharon)
¬ GrandDaugther(Tom, Bob)	Father(Sharon, Bob)
¬ GrandDaughter(Victor, Victor)	Father(Tom, Bob)
	Father(Bob, Victor)

FOIL Example

Example

GrandDaughter(x,y) $\leftarrow$

Candidate additions to precondition:
Equal(x,y), Female(x), Female(y), Father(x,y), Father(y,x), Father(x,z), Father(z,x), Father(z,y), and the negations to these literals

Assume greedy selection of Father(y,z):
GrandDaughter(x,y) $\leftarrow$ **Father(y,z)**

Candidate additions:
the ones from above and Female(z), Equal(z,y), Father(z,w), Father(w,z), and their negations

Learned Rule:
GrandDaughter(x,y) $\leftarrow$ **Father(y,z)** $\wedge$ **Father(z,x)** $\wedge$ **Female(y)**

FOIL Gain

$$FoilGain(L, R) \equiv t \left(log_2 \frac{p_1}{p_1 + n_1} - log_2 \frac{p_0}{p_0 + n_0} \right)$$

with

- L as new literal introduced in rule R to gain new rule R'
- t as number of positive bindings of rule R which are still covered by R'
- p_1 as number of positive bindings of rule R' and n_1 as number of negative bindings
- p_0 as number of positive bindings of rule R and n_0 as number of negative bindings

Remark: Bindings are the number of instantiations of the variables by constants. A binding is positive if the instantiated rule covers a positive example.

Learning Recursive Rule Sets

- Extend FOIL such that the target predicate can also be included in the preconditions with the same restrictions to variables as before.
- Problem: rule sets that produce infinite recursions

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

FFOIL

- FOIL learns rules which only work if the goal predicate is fully instantiated.
- Reason: FOIL learns **classification** rules for structured representations
- For IP you want that an output is **generated** not only checked
- Modification of FOIL for learning functions:
  - last argument introduced with a special role
  - bindings of partial clauses are defined as: positive, negative, undetermined
  - Gain calculation modified: each undetermined binding counts as 1 positive and $r - 1$ negative bindings with r as number of constants given in the examples
  - negative examples not necessary
- FFOIL is designed for and limited to the learning of functional relations

FFOIL

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

Example: Plus

Examples:
`plus(0,0,0), plus(1,0,1), plus(2,0,2), plus(0,1,1), plus(1,1,2), plus(0,2,2)`
Background Knowledge: `dec(1,0), dec(2,1)`

Start: `plus(A,B,C) :- .`
Bindings: $\langle 0,0,-\rangle 0$, $\langle 1,0,-\rangle 0$, $\langle 2,0,-\rangle 0$, $\langle 0,1,-\rangle 0$, $\langle 1,1,-\rangle 0$, $\langle 0,2,-\rangle 0$
Extended to: `plus(A,B,C) :- A=C.`
Bindings: $\langle 0,0,0\rangle +$, $\langle 1,0,1\rangle +$, $\langle 2,0,2\rangle +$, $\langle 0,1,1\rangle -$, $\langle 1,1,1\rangle -$, $\langle 0,2,0\rangle -$
Extended to: `plus(A,B,C) :- A=C, B=0.`
corresponds to `plus(A,0,A) :- !.`

Remaining Bindings: $\langle 0,1,-\rangle 0$, $\langle 1,1,-\rangle 0$, $\langle 0,2,-\rangle 0$
Start new rule: `plus(A,B,C) :- dec(B,D), dec(E,A).`
Extended Bindings: $\langle 0,1,-,0,1\rangle 0$, $\langle 1,1,-,0,2\rangle 0$, $\langle 0,2,-,1,1\rangle 0$
Finally: `plus(A,B,C) :- dec(B,D), dec(E,A), plus(E,D,C), !.`
Bindings: $\langle 0,1,1,0,1\rangle +$, $\langle 1,1,2,0,2\rangle +$, $\langle 0,2,2,1,1\rangle +$

What else?

- Omitting example-driven and schema-based ILP
- Functional-logic inductive programming FLIP (Hernandez-Orallo)
- Use of systematic enumeration instead of evolutionary methods, learn HASKELL programs with higher-oder functions: MAGICHASKELLER (Katayama)
- Specialized methods for specific applications: programming by demonstration