Automatic Inductive Programming

Ricardo Aler Mur

Universidad Carlos III de Madrid

http://www.uc3m.es/uc3m/dpto/INF/aler

http://et.evannai.inf.uc3m.es

Contents

Aims of the Tutorial

1. Survey of AIP: Automatic Inductive Programming is a fragmented field (ILP, GP, Program Synthesis, ...)
2. To understand AIP as an extension to Machine Learning
3. Focus on search-based techniques (mostly evolutionary techniques)

INTRODUCTION TO AIP. A SURVEY

- Introduction
- Deductive Automatic Programming
- Synthesis of Functional Programs
- ILP for Program Synthesis

Automatic Programming

- Automatic Generation of Programs
- The user says **what** to do, the computer builds a program that does it
- Saying what to do must be easier than writing the program by hand

Related Fields

- Universal Planning
- Production Rule Systems (PRS)
- Reinforcement Learning (learning general strategies)
- Recurrent Neural Networks (sequences of executions)
- Learning classifier system (rule-based systems. Pittsburgh and Michigan approaches)
- Inductive Logic Programming (powerful relational language)
- …

Importance of AIP

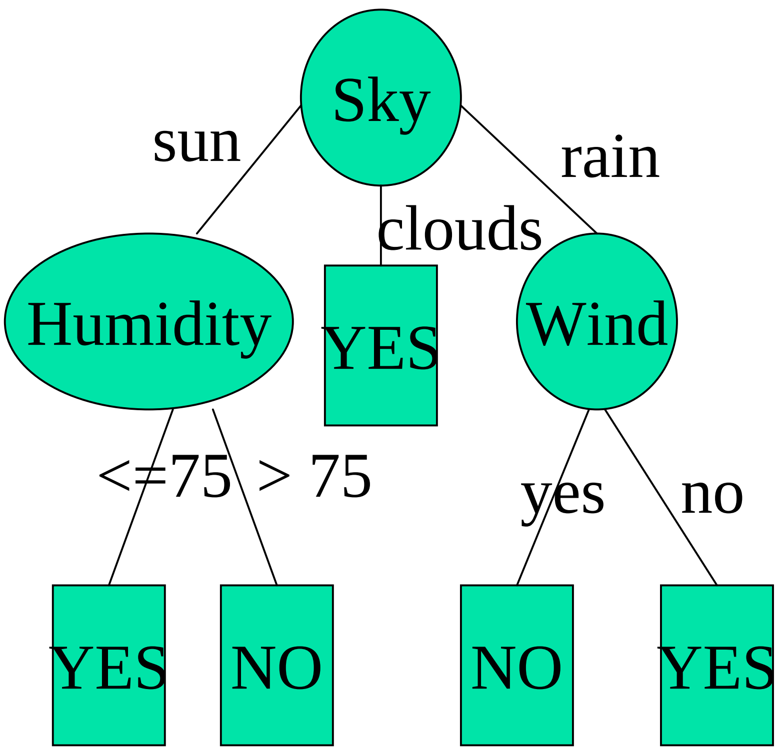
- From a scientific point of view:
  - A program is the most general structure that another program can learn (well beyond propositional Machine Learning)
- From a practical point of view:
  - There are problems whose solution is a computer program and not some other Machine Learning propositional structure (decision trees, neural networks, …)

Propositional Machine Learning. Input

Sky	Temperature	humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Clouds	83	86	No	Yes
Rain	70	96	No	No
Rain	68	80	No	Yes
Clouds	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Clouds	72	90	Yes	Yes

Propositional Machine Learning. Output

Decision trees



Rules

IF Sky = sun

Humidity <= 75 **THEN** Play = yes

ELSE IF Sky = sun

Humidity > 75 **THEN** Play = no

ELSE IF Sky = clouds **THEN** Play = yes

ELSE IF Sky = rain

Wind = Si **THEN** Play = yes

ELSE Play = no

Automatic Programming. Input (specification)

- Input/output pairs: (list sorting)
  - ([2,1], [1,2]); ([2,3,1], [1,2,3]);
  - ([3,5,4], [3,4,5]); ([],[]); ...
- Primitives:
  - *(dobl start end work):* for loop
  - *(wismaller x y):* return smaller
  - *(wibigger x y):* return bigger
  - *(swap x y)*
  - *(e1+ x) (e1- x) (e- x y)* : increase, decrease, substract

Automatic Programming. Output

```
(dobl (wismaller (wismaller (e1- *len*) *len*)
                  index)
      (dobl (wismaller index
                       (wismaller
                        (e1- index)
                        (e1+ (e1- index))))
            (e1- *len*)
            (swap (swap (e1- *len*) index)
                  index))
      (dobl (swap (wibigger index (e- index *len*))
                  (e- index *len*))
            (e1- *len*)
            (swap (wismaller (e1+ index) index)
                  index))))
```

Equivalente to (kind of "bubble sort")

```
(dobl 0
      (el- *len*)
      (dobl 0
            (el- *len*)
            (swap (wismaller (el+ index) index)
                  index))))
```

AIP as an Extension to Propositional Machine Learning

- Variable input size
- Use of conditionals (if-then-else, case)
- Reuse:
  - Use of variables (reuse of computations)
  - Use of subroutines (reuse of code)
  - Use of loops and recursivity (reuse of code)
- Turing-complete languages

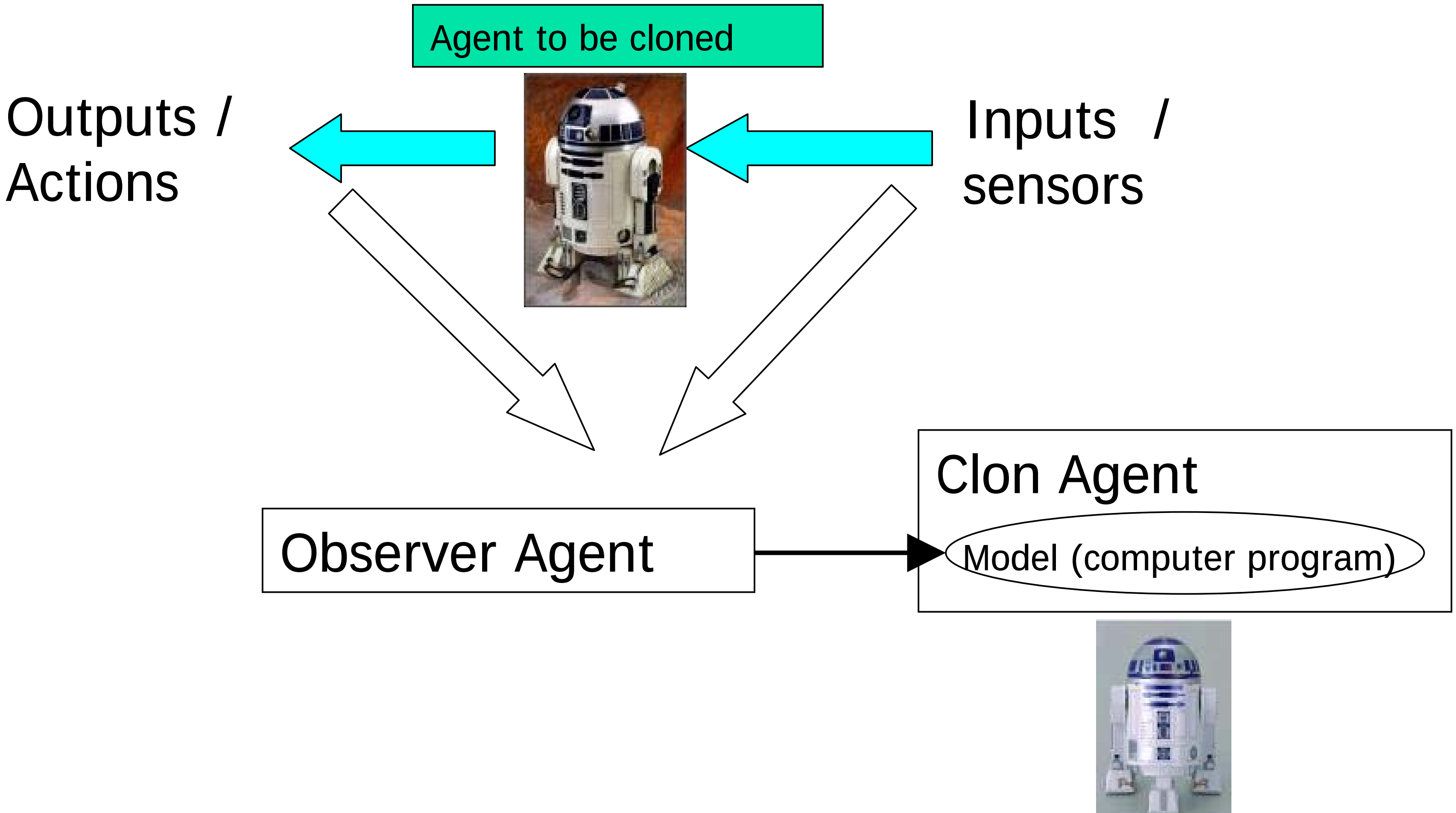
For What Kind of Problems?

- Complex domains where human beings find difficult to write programs
- And, full algorithms are required (with conditionals, subroutines, loops, …)

For What Kind of Problems?

- Programming quantum computers
- Programming parallel computers
- Machine Conde Programming
- Programming agents in complex domains (ej: Robosoccer)
- Programming text transformations from user supplied examples (web pages, ...)
- Behavioral cloning
- Etc.

Behavioral Cloning (Extracting Operational Knowledge)



Types of AIP

- **Deductive**: to generate a program from a high-level description
- **Inductive**: to generate a program from a set of instances

Deductive Automatic Programming

- Artificial Intelligence + Software Engineering
- **Main goal:** generate a program from a high-level description, easier (and shorter) to write than the actual program.
- However, this field includes compiler techniques for the optimisation of programs, tools for helping programmers, etc.

Deductive Automatic Programming Techniques

- Program analysis, transformation, and optimisation (compilers techniques)
  - Memoization, sentence ordering, tail recursivity, rewriting rules (* ?x 1) → ?x, ...
- Programming assistants (Apprentice)
- Scientific program generation (Kant)
- High-Level languages (SML, SETL –set theory based-)

Deductive Automatic Programming Tools

- Automatic Programming Server: http://www.cs.utexas.edu/users/novak/cgi/apdemo.cgi
  - Generating procedures for specialized types from abstract types
  - Type conversion
- Graphical Programming System: http://www.cs.utexas.edu/users/novak/cgi/gpserver.cgi

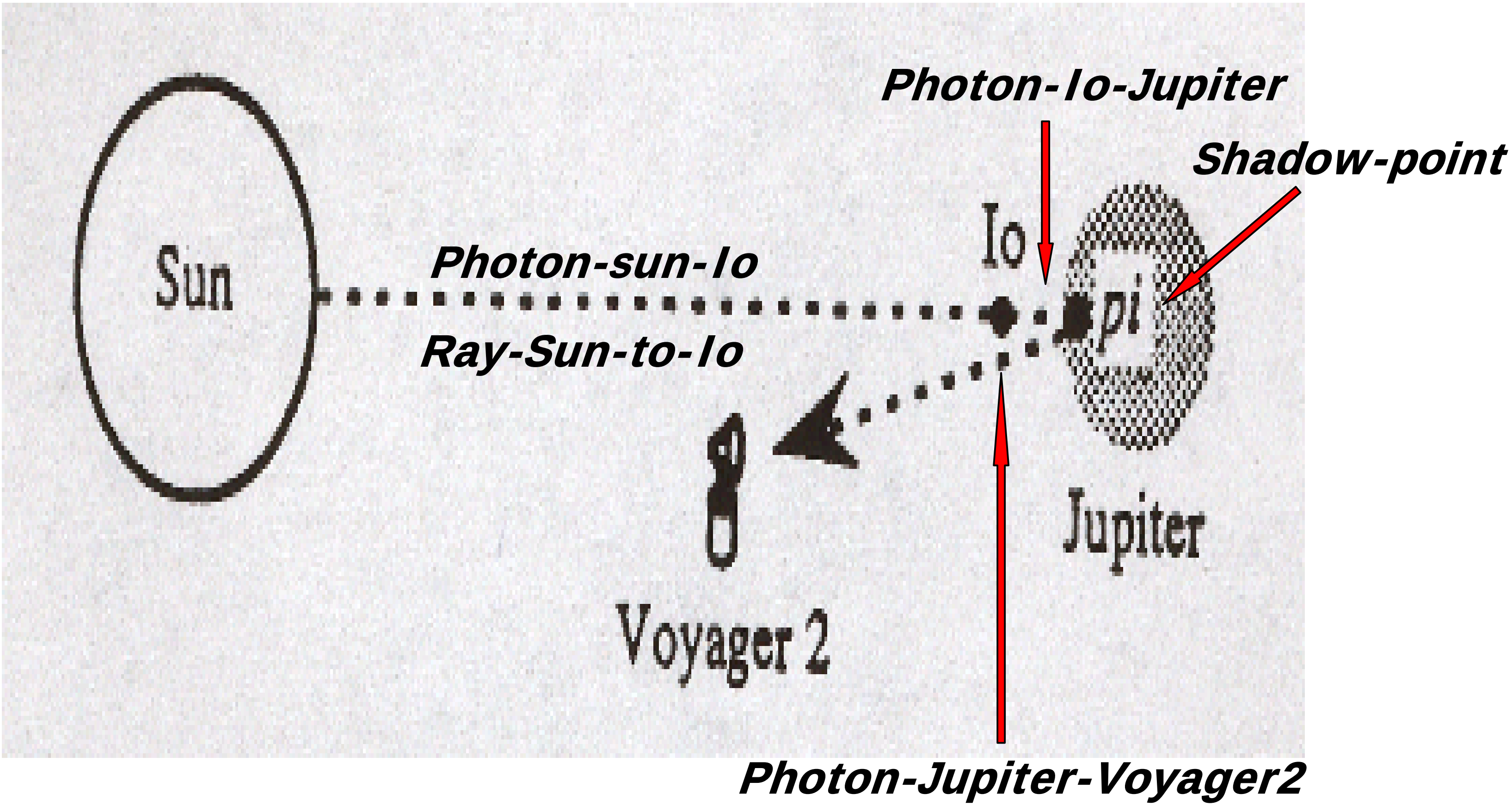
Deductive Automatic Programming

- Transformational and Deductive Systems (Refine, KIDS; Manna & Waldinger 92)
- Specifications are written by means of formal languages
- An specification is a theorem to prove
- An Automatic Theorem Prover constructs the program
- Specification **->** theorem **->** proof **->** program

Example: Amphion [Stickel, 95]

- “Deductive Composition of Astronomical Software from Subroutine Libraries”
- Astronomical domain (solar system)
- Example: generate a program that tells where the shadow of Io is on Jupiter at a particular time
- The program is made of calls to astronomical subroutines from the SPICE library

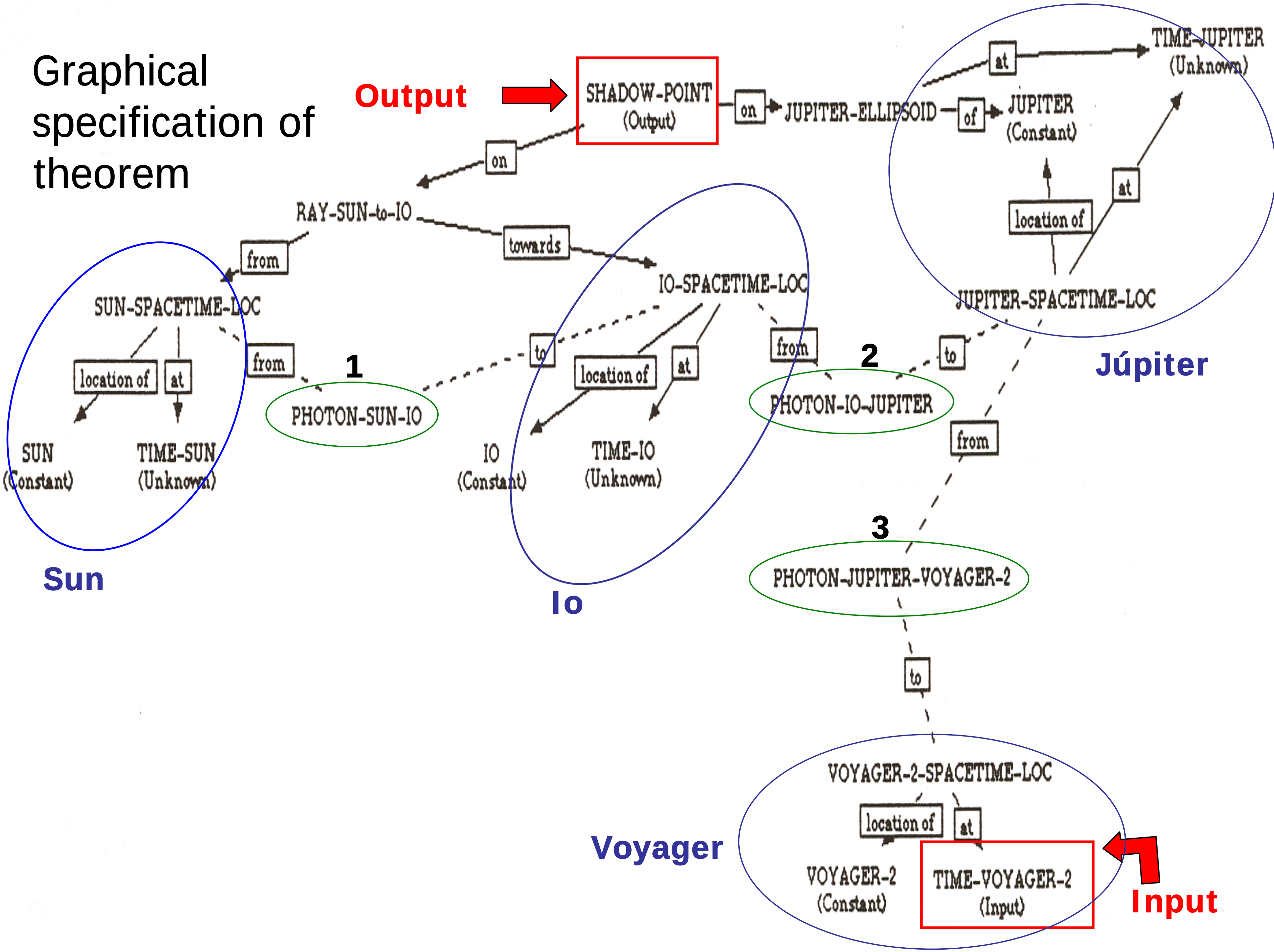
Where is the shadow of Io?



Amphion. Shadow of Io Theorem

- Is there a *shadow-point*, that is at the intersection of *Ray-Sun-to-Io* and *Júpiter-Ellipsoid* ?
- (exists **sp?**) in-ray(Sun,Io,Jupiter, Voyager, **sp**) & in-elipsoid(Jupiter, **sp**)
- This theorem is represented in graphical form
- Then converted to predicate logic
- Then a constructive proof is obtained by the SNARK theorem prover
- Then, a FORTRAN program is generated

Graphical specification of theorem



Theorem (first order logic)

```
(all (time-voyager-2-c)
      (find (shadow-point-c)
    (exists
    (time-sun sun-spacetime-loc time-io io-spacetime-loc
      time-jupiter jupiter-spacetime-loc time-voyager-2
      voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
      ray-sun-to-io)
    (and
     (= ray-sun-to-io
(two-points-to-ray
 (event-to-position sun-spacetime-loc)
 (event-to-position io-spacetime-loc)))
      (= jupiter-ellipsoid
(body-and-time-to-ellipsoid jupiter
     time-jupiter))
     (= shadow-point
(intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
      (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
      (lightlike? io-spacetime-loc jupiter-spacetime-loc)
      (lightlike? sun-spacetime-loc io-spacetime-loc)
      (= voyager-2-spacetime-loc
(ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
      (= jupiter-spacetime-loc
(ephemeris-object-and-time-to-event jupiter time-jupiter))
      (= io-spacetime-loc
(ephemeris-object-and-time-to-event io time-io))
      (= sun-spacetime-loc
(ephemeris-object-and-time-to-event sun time-sun))
      (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
      (= time-voyager-2
(abs ephemeris-time-to-time time-voyager-2-c))))))
```

FORTRAN PROGRAM

```
SUBROUTINE SHADOW ( TIMEVO, SHADOW )
DOUBLE PRECISION TIMEVO              ...
INTEGER JUPITE
PARAMETER (JUPITE = 599)             ...
DOUBLE PRECISION RADJUP ( 3 )        ...
CALL BODVAR ( JUPITE, 'RADII', DMYO, RADJUP )
TJUPIT = SENT ( JUPITE, VOYGR2, TIMEVO )
CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY20 )
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
TIO = SENT ( IO, JUPITE, TJUPIT )
CALL FINDPV ( IO, TIO, PIO, DMY30 )
TSUN = SENT ( SUN, 10, TIO )
CALL FINDPV ( SUN, TSUN, PSUN, DMY40 )
CALL VSUB ( PIO, PSUN, DPSPI )
CALL VSUB ( PSUN, PJUPIT, DPJPS )
CALL MXV ( MJUPIT, DPSPI, XDPSPI )
CALL MXV ( MJUPIT, DPJPS, XDPJPS )
CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ), RADJUP
              RADJUP ( 3 ), P, DMY90 )
CALL VSUB ( P, PJUPIT, DPJUPP )
CALL MTXV ( MJUPIT, DPJUPP, SHADOW )
END
```

Amphion

- Advantages:
  - Easy to use (after 1h training)
    - Experts: from 30m to 5m
    - Non-experts: from several days to 30m
- Lmitations:
  - Programs made of calls to subroutines, no conditionals, no loops, no recursivity

Pros/Cons of Deductive AP

- +: Generated programs are guaranteed to be **correct**
- - : In general, it is **difficult to write correct and complete formal especifications**, specially if the problem is not well-defined

Automatic Inductive Programming

- **Goal**: to generate computer programs from instances
- This is usually achieved by a heuristic search in the space of computer programs
- Pros/Cons:
  - +: Specifications are **easier to write**
  - -: Specifications are **not complete** -> It is not guaranteed that the generated program will be absolutelly correct

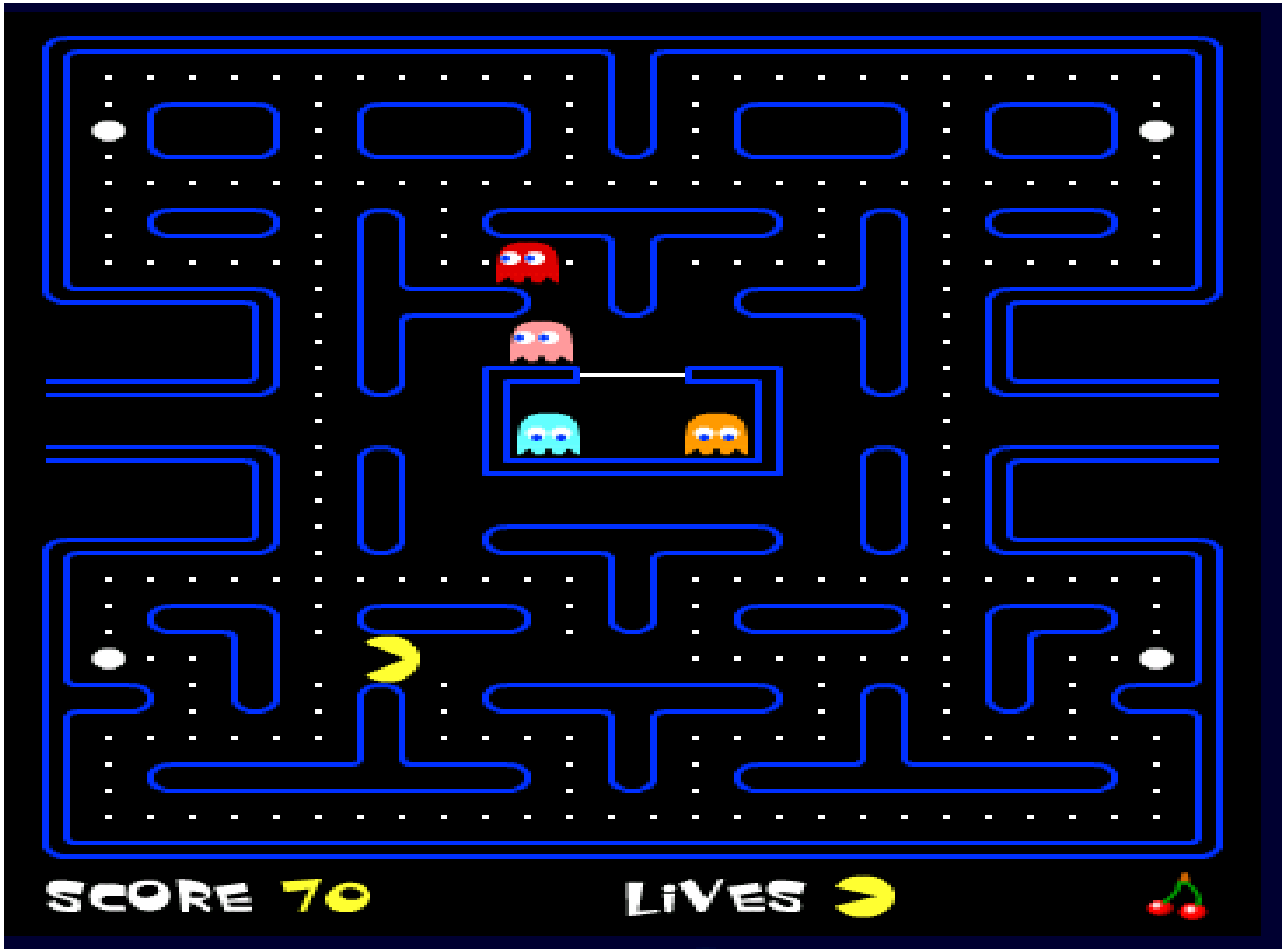
AIP specifications

- Specifications are composed of:
  - **Language**: primitives to be used by the AIP system to construct the solution program
  - **Heuristic**: evaluates candidate solutions (programs). There are basically, two types:
    - Input / Output pairs
    - Performance measure
    - (or combinations of both)

Input / Output specification

- Example: create a sorting program
- Input / output pairs:
  - ([2,1], [1,2]); ([2,3,1], [1,2,3]);
  - ([3,5,4], [3,4,5]); ([],[]); ...
- Primitives:
  - (dobl start end work) (wismaller x y)
  - (swap x y) (wibigger x y)
  - (e1+ x) (e- x y) (e1- x)

Performance Measure Specification



Performance measure specification

- Performance measure:
  - Count how many dots the Pacman ate in one game
- Primitives:
  - if-obstacle, if-dot, if-big-dot, if-phantom,
  - Forward, turn-left, turn-right

Example of strategy for Pacman

```
if-phantom then {
    turn-left;
    turn-left;
    go-forward;}
else if-big-dot {
      go-forward;
      girar-derecha;}
```

Types of Automatic Inductive Programming

- Synthesis-based: the program is built piece by piece, never actually executed
  - Synthesis of Functional Programs
  - Synthesis of Logic Programs
- Search-based:
  - A search technique (genetic algorithms, …) is used to search in the space of computer programs
  - Basically it is “iterated generate and test”
  - Candidate programs are executed (run) to determine how well they perform

LISP Program Synthesis

- **Seminal work:** Summers P. **1977**. "A Methodology for LISP Program Construction from Examples," *Journal of the ACM*
- Smith, D. 1984. “The Synthesis of LISP Programs from examples. A survey”. Mac Millan Publishing.

Two Steps in LISP Program Synthesis

1. Traces (computations) are created for individual input/output pairs
2. Then, patterns (like recurrence / recursivity) are identified in the traces

LISP Program Synthesis

- Idea: "For some classes of programs, a few well-chosen input/output pairs, determine the general program"
- Example (*last*): [(A),A]; [(A B), B]; [(A B C), C]
  - T_1: A=first((A))
  - T_2: B=first(rest ((A B)))
  - T_3: C=first(rest (rest ((A B C))))
  - T_K: last=first(rest … rest (list))
- General pattern (program): "Apply k-1 times *rest*, then apply *first*"

LISP Program Synthesis

- T_1: A=first((A))
- T_2: B=first(rest ((A B))) =
  - T_2 = T_1(rest((A B)))
- T_3: C=first(rest (rest ((A B C))))
  - T_3 = T_2(rest((A B C)))
- T_K: last=first(rest … rest (list))
  - T_k = T_{k-1}(rest(list))

LISP Program Synthesis

- That is, <u>traces</u> are obtained from input/output pairs, and then the general pattern is identified
- Actually, recursive programs are synthesized by applying Summers' Basic Synthesis Theorem

```
last(x) =
    Case singleton?(x)
        Yes: return first(x)
        No: return last(rest(x))
```

Structure of (Recursive) Learned Programs

- G(x) = F(x, constant)
- F(x,z) =

  Case

  p_1(x): f1(x,z)

  ...

  p_k(x): fk(x,z)

  Else H(x, F(b(x), G(x,z)))

X is the main variable, Z is a secondary variable

Examples of Programs

```
(last x) =
    (cond
        ((atom (cdr x))
            (car x))
        (T (last (cdr x)))))
```

```
(but-last x) =
        (cond
            ((atom (cdr x)) nil)
            (T (cons (car x)
                    (but-last (cdr
              x)))))
```

```
(reverse x) = (rev x ‘())
(rev x z) =
    (cond  ((atom x) z)
            (T (rev (cdr x) (cons (car x) z))))
```

Assumption on Input/Output Pairs

- *x/y* are sorted from simple to complex
- No atom (element) appears twice in *x*
- All atoms (elements) in *y* are also in *x* (selfcontained).
- If all this happens, each input/output pair has a **unique trace**
- Traces can be found by enumeration

LISP sublanguage

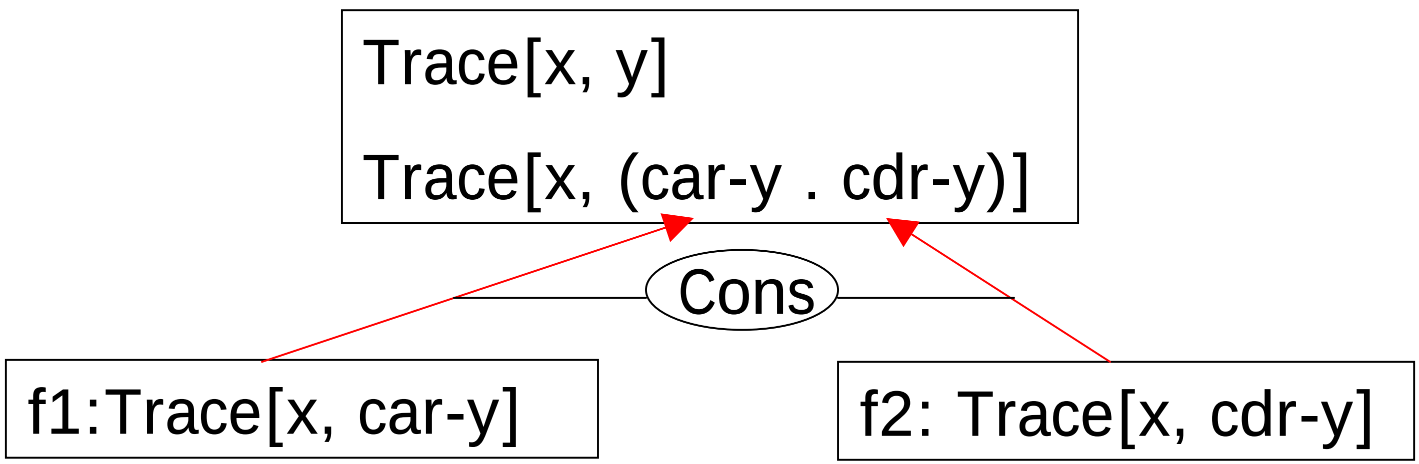
- Language:
  - Car: (car ‘(a b c)) = a
  - Cdr: (cdr ‘(a b c)) = (b c)
  - Cons: (cons ‘a ‘(b c)) = (a b c)
  - Atom: (atom ‘a) = T
  - Cond: conditional
- Operates only with lists (no numbers)

Obtaining traces

- For every input/output pair *(x,y),* find *f* such that:
  - $y = f(x)$

1. By enumeration of compositions of *car* and *cdr*, try to find a direct relation between *x* and *y*:
   - Ej: trace[(A),A] : *y = (car x)*
2. If that fails, then divide and conquer: find traces f_1 and f_2 such that:
   - $(car\ y) = f_1(x)$
   - $(cdr\ y) = f_2(x)$
   - $Trace[x,y]= (cons\ f_1(x)\ f_2(x))$

Obtaining Traces (Divide and Conquer)



Obtaining Traces

- No direct relation for Trace [(A B), (B)]
- X = (A B); Y = (B); (car Y) = B; (cdr Y) = ()
- Divide and conquer:
  - f1: (car Y) = B = (car (cdr X)
  - f2: (cdr Y) = () = ()
  - Trace[(A B), (B)] = (cons f1(X) f2(X)) = (cons (car (cdr X)) ())

Recurrence Detection (Basic Synthesis Theorem)

- **Let traces be:**
  - $y_1 = f_1(x_1)$
  - $y_2 = f_2(x_2)$
  - ...
- **If:** $\forall i \;\; f_{i+1}(x) = H(f_i(b(x)),x)$
  - $f_i(b(x))$ appears just once in H (b made of car/cdr, H made of cons/car/cdr)
  - This means that, for instance, f_2 is **embedded** in f_1
  - Pattern matching algorithms
- **Then:** F(x) = case

  $p_1(x) : f_1(x)$

  Else:H(F(b(x)),x)

Conclusions. Synthesis of LISP programs

- [Summers, 77] Identifies recursivity by detecting a trace being embedded in another trace
- It works because:
  - Restricts input/output *(x,y)* pairs so that trace *f* is unique in *y = f(x)*
  - It restricts the target to be learned (one-argument recursive functions)
  - It works only on structural tasks on lists. Structural: the task only depends on the structure of the list, not on its content. Sorting is beyond its scope.
  - Trace generation is domain dependent! (lists)
- Good idea: using traces

Applications of Program Synthesis

- Learning by Demostration / Learning by Example
- Teacher – Student paradigm
- Computation traces come from users, working through graphical interactive interfaces
- Example: TELS learns text-editing macros from the user and generalizes them with loops and conditionals [Witten et al. 93]

References

- “Watch What I do. Programming by Demonstration” [Cypher, 93]
- “Your Wish is My Command. Programming by Example” [Lieberman, 01]

Extensions. Synthesis of Functional Programs

- Schmidt, U. and Wysotzki, F. (1998). **"Induction of Recursive Program Schemes"**. *ECML'98*
- Kitzelmann, E., Schmidt, U., Mühlpfordt, M., and Wysotzki, F. 2002. **"Inductive Synthesis of Functional Programs"**. *Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Joint International Conference*
- E. Kitzelmann, U. Schmid. 2005. **"An Explanation Based Generalization Approach to Inductive Synthesis of Functional Programs"**. *ICML'05*

Schmidt Approach

- It goes well beyond Summer's
- Language independent
- Multiple recursion
- Multiple arguments
- Linear, tail, and tree recursion
- Mostly, structural tasks (lists, trees, …)
- Learning recursive programs is basically equivalent to learning some kind of grammars
- Application: XSL transformations (traces generated by Genetic Programming)

Some results (EBG paper)

Total time

function	#expl	#eqs	times in sec
last	4	2	.003 / .001 / .004
init	4	1	.004 / .002 / .006
evenpos	7	2	.01 / .004 / .014
switch	6	1	.012 / .004 / .016
unpack	4	1	.003 / .002 / .005
lasts	10	2	.032 / .032 / .064
mult-lasts	11	3	.04 / .49 / .53
reverse	6	4	.031 / .036 / .067

Types of Automatic Inductive Programming

- Synthesis-based:
  - Synthesis of Functional Programs
  - **Synthesis of Logic Programs:**
    - **Without schemes**
    - **With schemes**
- Search-based

Inductive Logic Programming (ILP)

- Machine Learning framework for learning first-order logic expressions (horn clauses)
- ILP language is more expressive than typical propositional ML languages
- Actually, it is basically Turing-complete (computer programs can be written in it with recursivity and “subroutines”)
- However, it is mostly used for relational concept learning, not for program synthesis

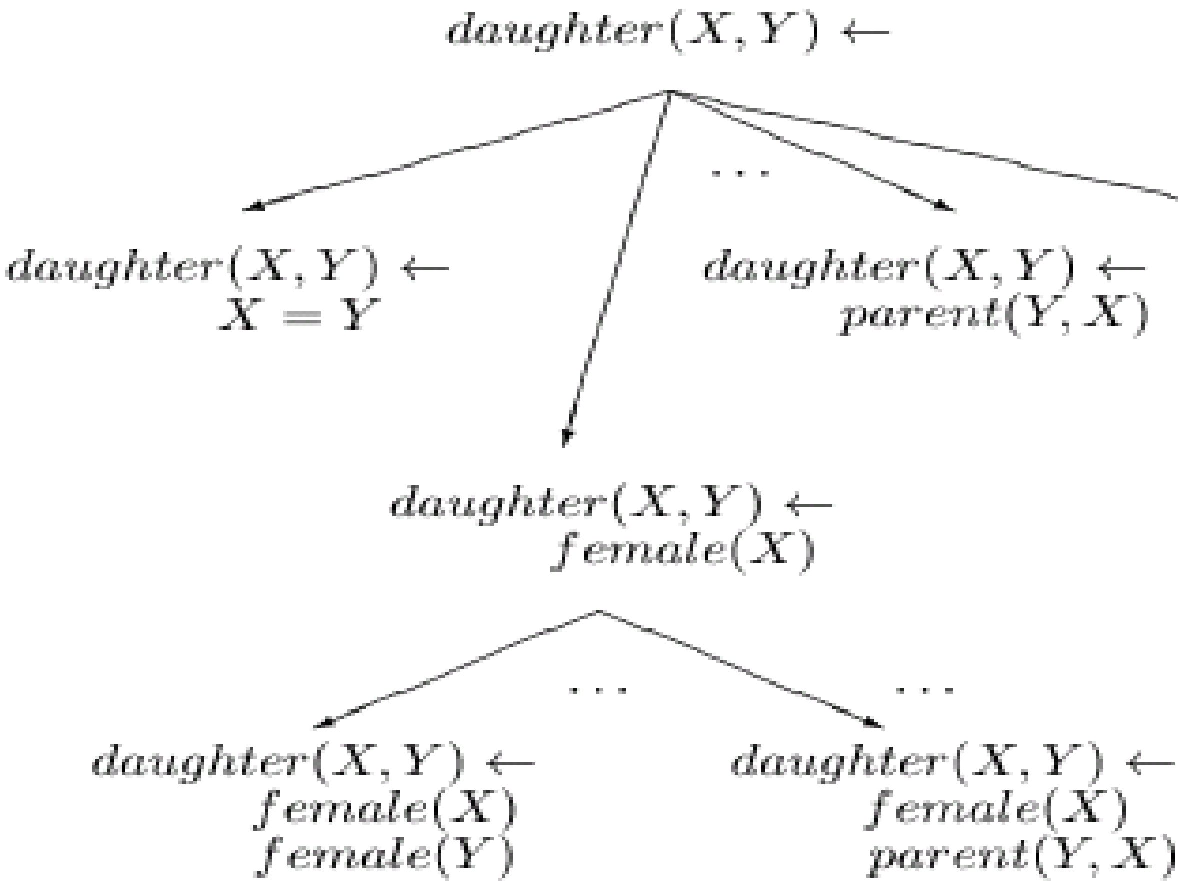
ILP Example

Training examples		*Background knowledge*	
$daughter(mary, ann).$	$\oplus$	$parent(ann, mary).$	$female(ann).$
$daughter(eve, tom).$	$\oplus$	$parent(ann, tom).$	$female(mary).$
$daughter(tom, ann).$	$\ominus$	$parent(tom, eve).$	$female(eve).$
$daughter(eve, ann).$	$\ominus$	$parent(tom, ian).$	

Learned Knowlege:

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$
$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

General to Specific (top-down) Search (FOIL)



- Also:
  - Bottom-up: GOLEM, CIGOL
  - Proposicional: LINUS

ILP Allows for Recursivity

Positive examples E^+:	Negative examples E^-:	Background knowledge
ancestor(akel, andrej).	ancestor(akel, akel).	parent(akel, andrej).
ancestor(andrej, boris).	ancestor(andrej, akel).	parent(andrej, boris).
ancestor(boris, miha).	ancestor(boris, andrej).	parent(boris, miha).
ancestor(akel, boris).	ancestor(boris, akel).	
ancestor(akel, miha).	ancestor(miha, boris).	
ancestor(andrej, miha).	ancestor(miha, akel).	

Obtained knowledge:

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

ILP for Program Synthesis

Positive and negative instances

subset([],[])
subset([],[a,b])
subset([d,c],[c,e,d])
subset([h,f,g],[f,i,g,h,j])

¬subset([k],[])
¬subset([n,m,m],[m,n])

Background knowledge: select(a, [2,a,3,4], [2,3,4])

select(X,[X|Xs],Xs) ←
select(X,[H|Ys],[H|Zs]) ← select(X,Ys,Zs)

Obtained program

subset([],Xs) ←
subset([X|Xs],Ys) ← select(X,Ys,Zs), subset(Xs,Zs)

Types of Synthesizers

- **No schemes**: TIM, MARKUS, SPECTRE, MERLIN, WIM, FILP, ...
- **With schemes**: SYNAPSE, DIALOG, METAINDUCE, CRUSTACEAN, CLIP, FORCE2, SIERES, ...
- Pierre Flener, Serap Yilmaz. 1999. **Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects.** *Journal of Logic Programming*
- Flener et al 1994. **ILP and Automatic Programming: Towards Three Approaches.** *4th International Workshop on ILP*

WIM (top-down, no schemes) [Popelinsky, 95]

	Number of positive examples	Number of hypotheses tested
member	2	3
concat	2	6
append	3	6
reverseConcat	2	5
reverseAppend	3	14
split	2	7
sublist	4	15
union	4	15
quicksort	7	2307

WiM **results if no assumption was needed**

5 seconds to 5 minutes

WIM. Results (1 query, interactive)

	Number of positive examples	Number of hypotheses tested
append	2	171
delete	2	21
last	2	99
plus	3	54
lessOrEqual	3	66
length	3	20
extractNth	3	27

WiM **results with 1 membership query**

Quicksort Code

```
; logical program for quicksort
qsort([X|Xs],Ys) :-
   partition(Xs,X,S1,S2),
   qsort(S1,S1s),
   qsort(S2,S2s),
   append(S1s,[X|S2s],Ys).
qsort([],[]).
```

But partition and append are primitives!

SYNAPSE (With Schemes) [Flener, 95]

- Critique: instances are weak specifications
- Goal:
  - Compress ([a,a,b,b,a,c,c,c], [a,2,b,2,a,1,c,3])
- From instances:
  - Compress ([],[])
  - Compress ([a], [a,1])
  - Compress ([b,b], [b,2])
  - Compress ([c,d], [c,1,d,1])
  - Compress ([e,e,e], [e,3])
  - Compress ([f,f,g], [f,2,g,1]
  - Compress ([j,k,l], [j,1,k,1,l,1])

SYNAPSE. Properties

- In addition to instances, it adds background knowledge in the form of properties:
  - Compress ([X], [X,1])
  - X=Y -> Compress([X,Y], [X,2])
  - X <> Y -> Compress([X,Y], [X,1,Y,1]
- Interactive

SYNAPSE. Schemes

- Divide and conquer:
- *R(X,Y) iff Minimal(X), Solve(Y)*
- *R(X,Y) ifff*
  - *1<= k <= c*
  - *Non-Minimal (X)*
  - *Decompose (X, HX, TX)*
  - *Discriminate*$_k$ *(HX,TX,Y)*
  - *R(TX,TY)*
  - *Process*$_k$ *(HX,HY)*
  - *Compose*$_k$ *(HY,TY,Y)*

SYNAPSE. Schemes

- (Divide and conquer simplified wrt [Flener, 95])
- *R(X,Y) iff Minimal(X), Solve(Y)*
- *R(X,Y) ifff*
  - *Non-Minimal (X)*
  - *Decompose X => Head + Tail*
  - *Solve(Head) => HY*
  - *Solve(Tail) => TY*
  - *Solution Y = HY + HX*

SYNAPSE. Algorithm

- Expansion phase:
  - Create a first approximation
  - Synthesis of *Minimal* and *Non-Minimal*
  - Synthesis of *Decompose*
  - Insertion of recursive atoms
- Reduction phase:
  - Synthesis of *Solve*
  - Synthesis of $Process_k$ and $Compose_k$

SYNAPSE. Other Solved Problems

- *Delete (E,L,R)* [6 instances, 3 properties]
- *Sort (L,S)* [10 instances, 1 property, *split, partition*]. Three programs:
  - *insertion-sort* $O(N^2)$,
  - *merge-sort* O(N log(N)),
  - *quicksort* O(N log(N)) were obtained by backtracking

ILP for Program Synthesis. Conclusions

- First order logic seems a very natural framework for learning programs (recursivity, “subroutines”, …)
- Formal approach
- Good idea: General-to-specific and specific-to-general search
- Good idea: schemes
- Simple programs can be learned

Other results on ILP for program synthesis

- J. Stahl. 1993. **“Predicate Invention in ILP – An Overview”**. *ECML*
- Hernández-Orallo, Ramírez-Quintana**.** 1999. **“Inductive Functional Logic Programming”**, *8th International Workshop on Functional and Logic Programming*
- Rao. 2005. **“Learning Recursive Prolog Programs with Local Variables from Examples”**. *ICML* (one-recursive)

Search Based AIP. General Idea

- **Incremental Search** in the space of computer programs. Generate and Test

Fac(n) = if ()
then {}

Else {}

Fac(n) = if (n=0)
then {1}

Else {n}

Fac(n) = if (n=0)
then {1}

Else {n*fac(n-1)}

Issues

- **1:** Search space is vast
- **2:** Programs are “fragile”. Recursive or iterative programs are even more fragile
  - => **How to transform programs?**
- **3:** An iterative or recursive program may never end (or take a long time)
  - **=> How to handle unlimited time?**
- **4:** No guarantee that the learned program is completely correct (induction)
  - => **How to handle many i/o pairs or long tests?**

Search-Based AIP

(Mostly, functional/procedural languages)

- Genetic Search (Genetic Programming):
  - Tree-based
  - Grammar-based
- Estimation of Distribution Algorithms:
  - Tree-based
  - Grammar-based
- Iterative Deepening: ADATE
- Other: Levin Search, Ant Colony Optimisation, ...

More general than synthesis-based but require a high computational effort!!