**The Logic Programming Paradigm**

- Logic programming: declarative, relational style.
- Origins linked to theorem proving and linguistics.
  - Try to solve problems / execute programs with a theorem prover.
  - Can you do that? Yes, but you (initially) end up with a sort of limited theorem prover.
  - Tabling also tries to overcome some of these initial limitations.
- Aim: clean, expressive, *obviously* correct programs.
Facts, Queries, Rules

person (mike).  person (alice).
person (bob).  person (mirna).

food (vegetable, salad).  likes (mike, vegetable).
food (vegetable, beans).  likes (mike, fish).
food (meat, chicken).  likes (alice, vegetable).
food (meat, rabbit).  likes (bob, fish).
food (meat, pork).  likes (bob, meat).
food (fish, soul).  likes (bob, vegetable).
food (fish, octopus).  likes (mirna, fish).
likes (mirna, vegetable).

meal (Person, Food): -
      likes (Person, Type),
      food (Type, Food).

same_food (P1, P2, Food): -
      meal (P1, Food),
      meal (P2, Food).

Execution of a Query

?- meal (alice, What).
likes (alice, T), food (T, What)

T = vegetables
What = salad
What = beans

Who = mike
T = vegetable

Who = mike
T = fish

Who = alice
T = vegetable

Who = bob
T = fish

Logical Variables

?- likes (mirna, X), X = meat.
no
The Logic Programming Paradigm

Arithmetic

- Prolog arithmetic is a compromise.
- \( X \) is \( \text{Exp} \)
  - Arithmetically evaluates \( \text{Exp} \)
  - Unifies result with \( X \)

?- \( X \) is 3 * sqrt(2).
  \( X = 4.242640687119286 \) ?

?- \( X = 3 \times \text{sqrt}(2) \).
  \( X = 3 \times \text{sqrt}(2) \) ?

?- 2 is 1 * X.
  \{ \text{ERROR: arithmetic:*/2, arg 1} - Instantiation Error \}

Interpretation of Formulas

- Rules and facts are stylized formulas:
  \( \text{meal} (\text{Person}, \text{Food}) : - \)
  \( \text{likes} (\text{Person}, \text{Type}), \text{food} (\text{Type}, \text{Food}) \).

- is a way of writing the formula
  \( \forall P \forall F \exists T \text{likes}(P, T) \land \text{food}(T, F) \rightarrow \text{meal}(P, F) \)

and a query

?- \( \text{meal} (\text{mirna}, X) \).

- is an attempt to mechanically and constructively determine whether

  \( \exists X. \text{meal}(\text{mirna}, X) \)

holds. If so, it is demonstrated by finding an \( X \) for which \( \text{meal}(\text{mirna}, X) \) can be inferred from the program. This is what a Prolog system does by applying an automated deduction procedure.

Structured Data

- \( \text{couple} (\text{married} (\text{mirna}, \text{bob}) ) \).

- \( \text{?- couple}(\text{C}) \).
  \( C = \text{married} (\text{mirna}, \text{bob}) \)

- \( \text{?- couple}(\text{married} (\text{mirna}, X)) \).
  \( X = \text{bob} \)

Lists

<table>
<thead>
<tr>
<th>What</th>
<th>Prolog notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty list</td>
<td>[]</td>
</tr>
<tr>
<td>Singleton list</td>
<td>[a]</td>
</tr>
<tr>
<td>Two-element list</td>
<td>[a, b]</td>
</tr>
<tr>
<td>List with a head a and a tail T</td>
<td>[a</td>
</tr>
<tr>
<td>List starting with a, b, c and continuing with tail T</td>
<td>[a, b, c</td>
</tr>
</tbody>
</table>

Pitfalls

- Redundant computations

  \( \text{hanoi}(N, L): - \)
  \( \text{hanoi}(N, A, B, C, L), A = a, B = b, C = c. \)

  \( \text{hanoi}(0, _, _, _, []). \)

  \( \text{hanoi}(N, A, B, C, M): - \)
  \( N > 0, \text{N1 is N - 1}, \text{hanoi}(\text{N1}, A, C, B, \text{M1}), \text{hanoi}(\text{N1}, C, B, A, \text{M2}), \text{append}(\text{M1}, [\text{move}(A, B)|\text{M2}], M). \)

  \( \text{fib}(0) := 0. \)
  \( \text{fib}(1) := 1. \)
  \( \text{fib}(N) := \text{fib}(N-1) + \text{fib}(N-2) \) \( : - N > 1. \)

- Non termination of otherwise correct programs

  \( \text{path}(A, B): - \)
  \( \text{edge}(A, C), \text{path}(C, B). \)
  \( \text{path}(A, B): - \text{edge}(A, B). \)
Tabling Basics

- Remember calls and their answers.
- Can improve speed.
- We will see two examples.

Towards Tabling: Memoing

- Solutions for a given size differ only on name of the pegs.
- There are two (recursive) calls of the same size.
  - Calling with free variables gives the most general answer.
- **Memoing** answer and recovering it adjusting variables in the answer to match those in the call avoids recomputing answer.
- Length of answer in order of computation steps, so constant speedup expected.
- Experimentally, 3-fold speedup by adding `- table hanoi/5`.

```prolog
hanoi(0, _, _, _, []).  % Base case
hanoi(N, A, B, C, M): -  % General case
    N > 0, N1 is N - 1,
    hanoi(N1, A, C, B, M1),
    hanoi(N1, C, B, A, M2),
    append([move(A, B)| M2], M).
```

<table>
<thead>
<tr>
<th>?- hanoi(2, A, B, C, L).</th>
</tr>
</thead>
<tbody>
<tr>
<td>L = [move(A,C),move(A,B),move(C,B)]</td>
</tr>
</tbody>
</table>

Towards Tabling: Memoing (Cont.)

```prolog
fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-1) + fib(N-2) :- N > 1.
```

<table>
<thead>
<tr>
<th>fib(0) := 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(1) := 1.</td>
</tr>
<tr>
<td>fib(N) := fib(N-1) + fib(N-2) :- N &gt; 1.</td>
</tr>
</tbody>
</table>

- Computing fib(N-1) includes the computation of fib(N-2).
- Memoing answers greatly speeds up computation.
- `- table fib/2.` reduces complexity from exponential \(O((\frac{1+\sqrt{5}}{2})^n)\) to linear!
**Towards Tabling: Termination Issues**

Logical reading:
- There is a path from $A$ to $B$ if they are directly connected by edges.
- There is a path from $A$ to $B$ if there is an edge from $A$ to $C$ and a connection from $A$ to $B$.

Question:
- Which nodes are reachable from $a$?

Standard Prolog (SLD) strategy leads to loops:

```
?- path(a, X).
edge(a, C₁), path(C₁, X)
edge(a, b), path(b, X)
edge(a, b), edge(b, C₂), path(C₂, X)
edge(a, b), edge(b, a), path(a, X)
```

- Initial reaction: carry around list of visited nodes.
- But: is it necessary? What is wrong with the program? Does not it precisely specify the meaning of being reachable?

**Towards Tabling: Bottom-Up Evaluation**

Read the program as logic:

```
path(A, B) ← edge(A, C) ∧ path(C, B).
path(A, B) ← edge(A, B).
edge(a, b). edge(b, a). edge(b, c).
```

From \{edge(a, b), edge(b, a), edge(b, c)\}
and path(A, B) ← edge(A, B)
infer \{path(a, b), path(b, a), path(b, c)\}
using path(A, B) ← edge(A, C) ∧ path(C, B)
infer \{path(a, a), path(b, b), path(a, c)\}

Nothing more can be inferred. The meaning of the program is:

\{edge(a, b), edge(b, a), edge(b, c), path(a, b), path(b, a), path(b, c), path(a, a), path(b, b), path(a, c)\}

Note that:
- We have not changed the program (no visited node list).
- But: we wanted to find out which nodes are reachable from node $a$ and we computed much more.
  - We actually computed the least fixpoint of the program (its standard semantics).
  - Other applications could require the greatest fixpoint.
- More on bottom-up evaluation surely in the Deductive Databases talk.
Towards Tabling: Suspending and Resuming

Challenge: a goal-directed (top-down) strategy which
- Detects loops.
- Derives only the consequences necessary for top-down execution.

Idea: when we have \( p : q, p, r \).
and the call to \( p \) is repeated (enters loop):
- \textit{Suspend} the computation before calling \( p \).
- Use the fact \( p \) to generate a solution to the original query.
- Use this solution to resume the suspended computation.

Some Applications

Tabled Evaluation

Example

:- table p/1.

\[
\begin{align*}
p(A) :&= q, \\
p(B), A &= 2. \\
p(A) :&= A = 1. \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Subgoal</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( p(A) )</td>
<td>1. ( A = 1 )</td>
</tr>
<tr>
<td>2. ( p(B), A = 2. )</td>
<td>8. ( A = 2 )</td>
</tr>
<tr>
<td>3. ( A = 1. )</td>
<td>13. Complete</td>
</tr>
</tbody>
</table>

Linear tabling uses recomputation instead of suspension.

Parsing

Let us assume input has been tokenized

<table>
<thead>
<tr>
<th>Arithmetic terms</th>
<th>Examples of strings we want to parse:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( expr \rightarrow expr + term )</td>
<td>String</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( expr \rightarrow term )</td>
<td>( 3 + 5 )</td>
</tr>
<tr>
<td>( term \rightarrow term * fact )</td>
<td>( 3 * 4 )</td>
</tr>
<tr>
<td>( term \rightarrow fact )</td>
<td>( 2 + 3 * 4 )</td>
</tr>
<tr>
<td>( fact \rightarrow ( expr ) )</td>
<td>( (2 + 3) * 4 )</td>
</tr>
<tr>
<td>( fact \rightarrow int(\text{Int}) )</td>
<td></td>
</tr>
</tbody>
</table>
Some Applications

Parsing (Cont.)

Attempt of a Prolog parser for the previous grammar

\[
\begin{align*}
\text{expr} & : \rightarrow \text{expr} + \text{term} | \text{term} \\
\text{term} & : \rightarrow \text{term} * \text{fact} | \text{fact} \\
\text{fact} & : \rightarrow (\text{expr}) | \text{int}(\text{Int})
\end{align*}
\]

\[
\text{expr}(S0, S) :- \text{expr}(S0, S1), S1 = [+| S2], \text{term}(S2, S).
\]

\[
\text{expr}(S0, S) :- \text{term}(S0, S).
\]

\[
\text{term}(S0, S) :- \text{term}(S0, S1), S1 = [*| S2], \text{fact}(S2, S).
\]

\[
\text{term}(S0, S) :- \text{fact}(S0, S).
\]

\[
\text{fact}(S0, S) :- S0 = ['('|S1], \text{expr}(S1, S2), S2 = [')'|S].
\]

\[
\text{fact}(S0, S) :- S0 = [N|S], \text{integer}(N).
\]

\[
\text{expr}(I, R) \text{ means: "} R \text{ is a suffix of } I \text{, and the prefix of } I \text{ w.r.t. } R \text{ is a valid expression."
}\]

?- \text{expr}([3 , +, 4, *], R).
R = [+] R = [+,4,*]

Complete parsing: \text{expr(Input, \[])}.

\footnote{Prolog provides a specific syntax to write parsers, closer to the "→" notation.}

Parsing (Cont.)

- A left-recursive rule

\[
\text{expr}(S0, S) :- \text{expr}(S0, S1), S1 = [+| S2], \text{term}(S2, S).
\]

generates a repeated computation and a loop.

- Tabling left-recursive rules amends this problem:

\[
\text{:- table expr/2, term/2.}
\]

Then:

?- \text{expr}([3, +, 4, *], []).
no

?- \text{expr}([3, +, 4, *, 7], []).
yes

?- \text{expr}(['(', 3, +, 4, ')', *, 7], []).
yes

?- E = [_,_,_,_,_,_,_], \text{expr}(E, []).
no

Tabling and Parsing

- Tabling + a recursive descendant parser (like the one we have presented) implements an Earley parsing:

  The Earley parser is a type of chart parser mainly used for parsing in computational linguistics, named after its inventor, Jay Earley. The algorithm uses dynamic programming. — Wikipedia

- Earley parsers parse all context-free grammars:
  - Complexity \(O(n^3)\) for ambiguous grammars.
  - Complexity \(O(n^2)\) for unambiguous grammars.
  - Linear for a large class of grammars.
Automatic Dynamic Programming

- Knap-sack problem: given $n$ items of integer size $k_i$ (1 ≤ $i$ ≤ $n$), and a knapsack size $K$:
  - Determine whether there is a subset of the items that sums to $K$.
  - Find such a subset — we are not maximizing / minimizing.
- Program: try all items ($n, n-1, \ldots, 1$) and decide, for each item, whether to include it or not.

ks(0,0).
ks(I,K) :- I >0, % Skip item Ith
  I is I-1, ks(I1,K).
ks(I,K) :- I >0, % Include item Ith
  item_size(I,Ki),
  K1 is K-Ki, K1 >= 0,
  I1 is I-1, ks(I1,K1).
item_size(1,2). item_size(2,3).
item_size(3,5). item_size(4,6).

- Worst-case complexity is $2^n$.

Knapsack and (again) Graph Traversal

- Decision at every node vs. incrementally building a transitive closure.
- In fact, Knap-sack solution very similar to graph traversal.
- Many problems very similar to graph traversals — or directly graph traversal.
  - E.g., model checking.
Tabling: Recap and Big Picture

- Tabling terminates and is complete for all calls with the bounded-term-depth property (those for which there is a bound on the size of the terms which can be generated).
- Tabling computes a fix point (the least fix point) for tabled predicates in a goal-driven way.
- Improves efficiency (automatic dynamic programming).
- Improves termination.

Advanced Tabling Capabilities
Variant vs. Subsumption Tabling

- $p(f(Y),X,1)$ and $p(f(Z),U,1)$ are variants as one can be made to look like the other by a renaming of the variables.

- Let us have $t_1: p(f(Y),X,1)$ and $t_2: p(f(Z),Z,1)$.
  - We can rename variables in $t_1$ to become $t_2$, but not the other way around.
  - They are not variants.
  - $t_1$ subsumes (is more general than) $t_2$.

- Call Variance vs Call Subsumption:
  - Either Variance or Subsumption can be used to check if we are executing a new tabled call.

- Answer Variance vs Answer Subsumption:
  - Either Variance or Subsumption can be used to check if we have computed a new answer for a tabled predicate.

Variant Tabling:
- More efficient table look-up.
- Better for side-effects.
- Useful for goal-directed queries and meta-interpreters.

Subsumption Tabling:
- Can avoid more recomputation (catches more similarities).
- Better termination:
  
  ```
  :- table ps/1 as subsumptive.
  p(X) :- p(s(X)).
  ```
- Can save space in internal tables.

Subsumption can be parametric

- It is natural to use syntactical subsumption.

- However, from a more semantic / knowledge reasoning point of view, terms may have an interpretation from which subsumption can be inferred.

- E.g., the term $X > 3$ subsumes $X = 4$ in the arithmetic domain.

- $is(some\_animal, dog)$ is subsumed by $belongs(some\_animal, mammal)$.

- Lattice for subsumption check.

- Also in abstract interpretation: relations in lattice of abstract domain encoded in subsumption lattice.
  - Remember tabling computes a fixpoint.
  - Basically, what an abstract interpreter does! (and the mechanism is not very different)

Negation and Well-Founded Semantics

Gives semantics for all normal logic programs. It allows logic programs to adequately handle inconsistencies and paraconsistencies, for example:

"The village barber shaves everyone in the village who does not shave himself."

```
shaves(barber, Person):-
  villager(Person),
  not shaves(Person, Person).
shaves(doctor, doctor). % Play with this
```

```

villager(barber). villager(doctor). villager(mayor).

?- shaves(X, Y).
```

- Applications: verification, Flora-2, medical informatics...
Suspenion Based Implementation

- Sharing local stack and heap.
- Copying those bindings which are different for each consumer.

```
POINTS
CHOICE
G
TRAIL
C1
HEAP
...
[1
2
p
q]
```

```
Carro, Chico de Guzmán (IMDEA, UPM)
Tabling and applications
Prometidos SS 42 / 45
Carro, Chico de Guzmán (IMDEA, UPM)
Tabling and applications
Prometidos SS 42 / 45
```
Suspension Based Implementation
- Sharing local stack and heap.
- Copying those bindings which are different for each consumer.
Suspension Based Implementation

- Sharing local stack and heap.
- Copying those bindings which are different for each consumer.

Open Research Topics

- Pruning of Answer-On-Demand Tabling.
- Subsumptive tabling with constraints.
- Pruning of previous subsumed calls.
- Call abstraction.
- Parallelism.
- Table compression.
- Side-effects.
Thanks

Parts of this presentation draw from material from:

- Terrance Swift (CENTRIA, Universidade Nova de Lisboa)
- David S. Warren (NYU at Stony Brook)

to whom we thank.