Tabled Logic Programming and Its Applications

Manuel Carro[1,2] Pablo Chico de Guzmán[1]

[1]School of Computer Science, Technical University of Madrid, Spain
[2]IMDEA Software Institute, Spain

Prometidos-CM Summer School
Facultad de Informática, UCM – September 19-21, 2011

Roadmap

- Introduction: the Logic Programming paradigm.
- Pitfalls.
- Tabling Basics:
  - Memoing.
  - Breaking loops.
- Analysis of some applications.
- Advanced Tabling Capabilities:
  - Subsumptive tabling.
  - Negation with tabling.
- Implementation issues.
- Open research topics.

The Logic Programming Paradigm

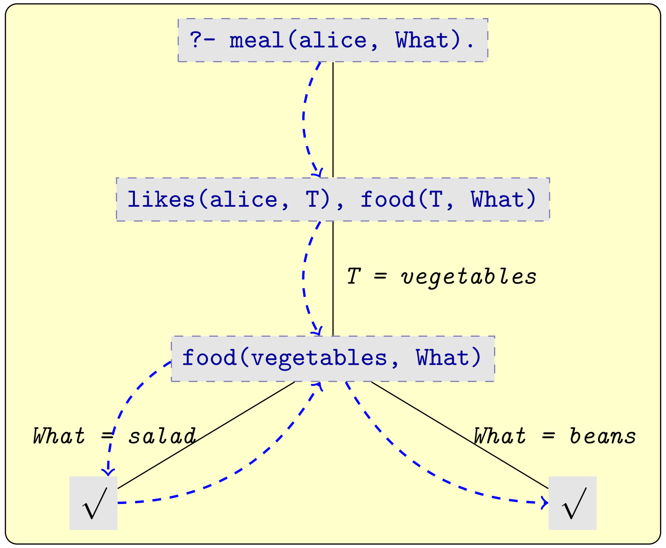
Intro to LP

- 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.

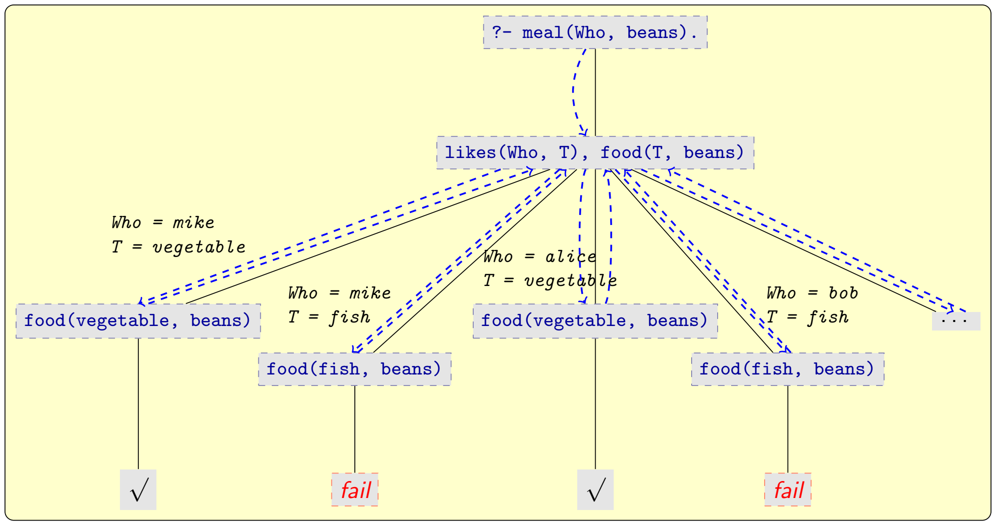
Facts, Queries, Rules

```
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



Execution of a Query



Logical Variables

```
?- likes(mirna, X), X = meat.
no
```

Arithmetic

- Prolog arithmetic is a compromise.
- `X is Exp`
  - Aritmetically evaluates `Exp`
  - Unifies result with `X`

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

?- X = 3 * sqrt(2).
   X = 3*sqrt(2) ?

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


Interpretation of Formulas

Rules and facts are stylized formulas:

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

is a way of writing the formula

$$\forall P\ \forall F\ \exists T\ likes(P, T) \wedge food(T, F) \rightarrow meal(P, F)$$

and a query

```
?- meal(mirna, X).
```

is an attempt to mechanically and constructively determine whether

$$\exists X.meal(mirna, X)$$

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

Structured Data

```
couple(married(mirna, bob)).

?- couple(C).
   C = married(mirna, bob)

?- couple(married(mirna, X)).
   X = bob
```

Lists

What	Prolog notation
Empty list	`[]`
Singleton list	`[a]`
Two-element list	`[a, b]`
List with a head `a` and a tail `T`	`[a|T]`
List starting with `a`, `b`, `c` and continuing with tail `T`	`[a, b, c|T]`

Pitfalls

Redundant computations

```
hanoi(N, L):-
    hanoi(N, A, B, C, L),
    A = a, B = b, C = c.

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

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

Non termination of otherwise correct programs

```
path(A, B):-
    edge(A, C),
    path(C, B).
path(A, B):- edge(A, B).
```

Tabling Basics

Towards Tabling: Memoing

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

Towards Tabling: Memoing (Cont.)

```
hanoi(0, _A, _B, _C, []).
hanoi(N, A, B, C, M):-
    N > 0, N1 is N - 1,
    hanoi(N1, A, C, B, M1),
    hanoi(N1, C, B, A, M2),
    append(M1, [move(A, B)|M2], M).

| ?- hanoi(2, A, B, C, L).
     L = [move(A,C),move(A,B),move(C,B)]
```

- 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`.

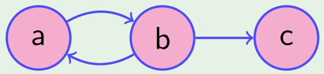
Towards Tabling: Memoing (Cont.)

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

- 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

```
path(A, B):- edge(A, C), path(C, B).
path(A, B):- edge(A, B).

edge(a, b). edge(b, a). edge(b, c).
```



Logical reading:

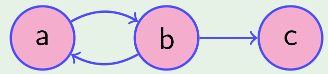
- There is a path from *A* to *B* if they are directly connected by edges.
- There is a path from *A* to *B* if there is an edge from *A* to *C* and a connection from *A* to *B*.

Question:

- Which nodes are reachable from *a*?
  `?- path(a, X).`

Towards Tabling: Termination Issues

```
path(A, B):- edge(A, C), path(C, B).
path(A, B):- edge(A, B).

edge(a, b). edge(b, a). edge(b, c).
```



Standard Prolog (SLD) strategy leads to loops:

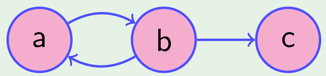
```
?- path(a, X).
      edge(a, C1), path(C1, X)
      edge(a, b), path(b, X)
      edge(a, b), edge(b, C2), path(C2, X)
      edge(a, b), edge(b, a), path(a, X)
```

- 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).
```



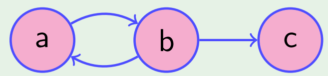
From	{edge(a, b), edge(b, a), edge(b, c)}
and	path(A, B) ← edge(A, B)
infer	{path(a, b), path(b, a), path(b, c)}
using	path(A, B) ← edge(A, C) ∧ path(C, B)
infer	{path(a, a), path(b, b), path(a, c)}

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)}

Towards Tabling: Bottom-Up Evaluation

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

edge(a, b). edge(b, a). edge(b, c).
```

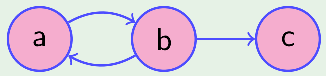


Note that:

- 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

```
path(A, B) :- edge(A, C), path(C, B).
path(A, B) :- edge(A, B).

edge(a, b). edge(b, a). edge(b, c).
```



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.
p.
```

and the call to p is repeated (enters loop):

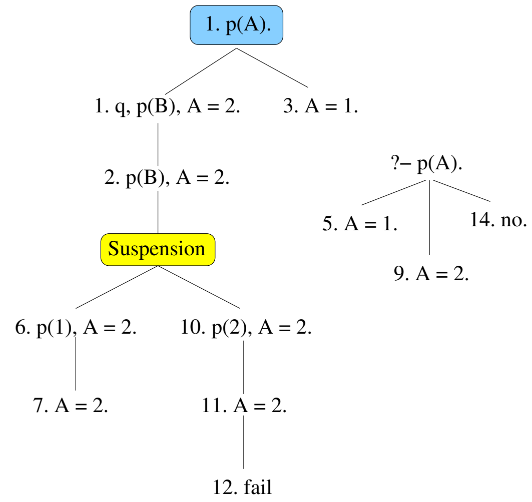
1. *Suspend* the computation before calling p.
2. Use the fact p to generate a solution to the original query.
3. Use this solution to resume the suspended computation.

Tabled Evaluation

Example

```
:- table p/1.

p(A) :-
    q,
    p(B),
    A = 2.

p(A) :-
    A = 1.
```

Subgoal	Answers
1. p(A)	4. A = 1 8. A = 2 13. Complete



- Linear tabling uses recomputation instead of suspension.

Some Applications

Parsing

Let us assume input has been tokenized

Arithmetic terms

```
expr  ⟶  expr + term
expr  ⟶  term
term  ⟶  term * fact
term  ⟶  fact
fact  ⟶  ( expr )
fact  ⟶  int(Int)
```

Examples of strings we want to parse:

String	**Evaluates to**
3	3
3 + 5	8
3 * 4	12
2 + 3 * 4	14
(2 + 3) * 4	20

Parsing (Cont.)

Attempt of a Prolog parser for the previous grammar[1]

```
expr  ⟶  expr + term | term
term  ⟶  term * fact | fact
fact  ⟶  ( expr ) | int(Int)
```

```
expr(S0,S) :- expr(S0,S1), S1 = [+|S2], term(S2,S).
expr(S0,S) :- term(S0,S).
term(S0,S) :- term(S0,S1), S1 = [*|S2], fact(S2,S).
term(S0,S) :- fact(S0,S).
fact(S0,S) :- S0 = ['('|S1], expr(S1,S2), S2 = [')'|S].
fact(S0,S) :- S0 = [N|S], integer(N).
```

expr(I, R) means: "R is a suffix of I, and the prefix of I w.r.t. R is a valid expression."

```
?- expr([3, +, 4, *], R).
R = [*]
R = [+,4,*]
```

Complete parsing: ?- expr(Input, []).

[1] Prolog provides a specific syntax to write parsers, closer to the "⟶" notation.

Parsing (Cont.)

Parsing in action

```
expr(S0,S) :- expr(S0,S1), S1 = [+|S2], term(S2,S).
expr(S0,S) :- term(S0,S).
term(S0,S) :- term(S0,S1), S1 = [*|S2], fact(S2,S).
term(S0,S) :- fact(S0,S).
fact(S0,S) :- S0 = ['('|S1], expr(S1,S2), S2 = [')'|S].
fact(S0,S) :- S0 = [N|S], integer(N).
```

?- expr([3], []). (wait forever...)

What is happening?

```
?- expr([3], []).
expr([3], S1₁), S1₁ = [+|S2₁], term(S2₁, [])
expr([3], S1₂), S1₂ = [+|S2₂], term(S2₂, S1₁), S1₁ = [+|S2₁], ...
```


Parsing (Cont.)

- A left-recursive rule

  ```
  expr(S0,S) :- expr(S0,S1), S1 = [+|S2], term(S2,S).
  ```

  generates a repeated computation and a loop.
- Tabling left-recursive rules amends this problem:

  ```
  :- table expr/2, term/2.
  ```

- Then:

  ```
  ?- expr([3, +, 4, *], []).
  no
  ?- expr([3, +, 4, *, 7], []).
  yes
  ?- expr(['(', 3, +, 4, ')', *, 7], []).
  yes
  ?- E = [_,_,_,_,_,_,_], 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.

For Completeness: Earley Parser

```
function EARLEY-PARSE(words, grammar)
  ENQUEUE((γ → •S, [0,0]), chart[0])
    for i ← from 0 to LENGTH(words) do
        for each state in chart[i] do
          if INCOMPLETE?(state) then
              if NEXT-CAT(state) is a nonterminal then
                  PREDICTOR(state)          // non-terminal
              else do
                  SCANNER(state)            // terminal
          else do
              COMPLETER(state)
        end
    end
    return chart

procedure PREDICTOR((A → α•B, [i, j])),
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → •γ, [j, j]), chart[ j])
    end

procedure SCANNER((A → α•B, [i, j])),
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
        ENQUEUE((B → word[j], [j, j + 1]), chart[j + 1])
    end

procedure COMPLETER((B → γ•, [ j, k])),
    for each (A → α•Bβ, [i, j]) in chart([j]) do
        ENQUEUE((A → αB•β, [i, k]), chart[k])
    end
```


Automatic Dynamic Programming

- Knap-sack problem: given n items of integer size k_i ($1 \leq i \leq n$), and a knap-sack 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
           I1 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.

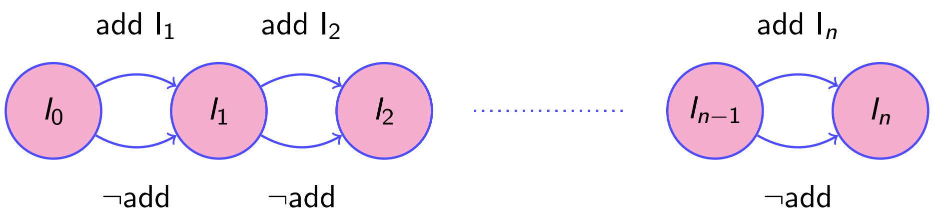
Automatic Dynamic Programming

- A tabling solution to the knapsack problem:

```
:- table ks/2.
ks(0,0).
ks(I,K) :- I>0,
           I1 is I-1, ks(I1,K).
ks(I,K) :- I>0,
           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 $O(nK)$.
- Intuitition: for every n and every number smaller than or equal to K, we check only once whether it succeeds or fails.

Knapsack and (again) Graph Traversal



- Decision at every node vs. incrementally building a transitive closure.
- In fact, Knapsack solution very similar to graph traversal.
- **Many** problems very similar to graph traversals — or directly graph traversal.
  - E.g., model checking.

For Completeness: Udi Manber Procedural Version

```
Algorithm Knapsack (S, K) ;
Input: S (an array of size n storing the sizes of the items),
    and K (the size of the knapsack).
Output: P (a two-dimensional array such that P [i, k ].exist = true if there
    exists a solution to the knapsack problem with the first i elements and a
    knapsack of size k, and P [i, k ].belong = true if the ith element belongs
    to that solution).
    { See Exercise 5.15 for suggestions about improving this program. }

begin
    P [0, 0].exist := true ;
    for k := 1 to K do
        P [0, k ].exist := false ;
    { there is no need to initialize P [i, 0] for i ≥ 1, because it will
        be computed from P [0, 0] }
    for i := 1 to n do
        for k := 0 to K do
            P [i, k ].exist := false ; { the default value }
            if P [i – 1, k ].exist then
                P [i, k ].exist := true ;
                P [i, k ].belong := false
            else if k – S [i ] ≥ 0 then
                    if P [i – 1, k – S [i ]].exist then
                        P [i, k ].exist := true ;
                        P [i, k ].belong := true
end
```


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.

Tabling: Recap and Big Picture

- It is not complete (naturally) for programs with an infinite least fixpoint:

```
n(0).
n(s(X)):- n(X).
?- n(X), X = a.
```

- It terminates for programs which loop due to repeated calls.

```
n(s(X)):- n(X).
?- n(X).
```

- It may not terminate for programs without bound term depth property:

```
p(X):- p(s(X)).
?- p(X).
```


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 vs. Subsumption Tabling (Cont.)

- 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.

Variant vs. Subsumption Tabling (Cont.)

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."

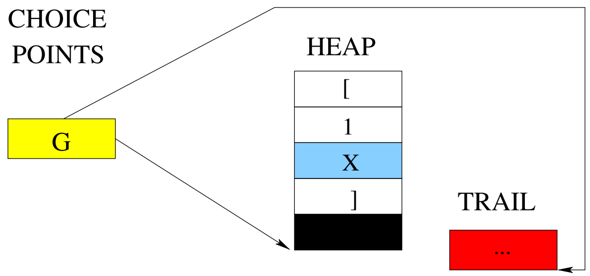
```
shaves(barber,Person):-
      villager(Person),
      not shaves(Person,Person).
shaves(doctor,doctor).  % Play with this

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

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

- Applications: verification, Flora-2, medical informatics...

Implementation Break

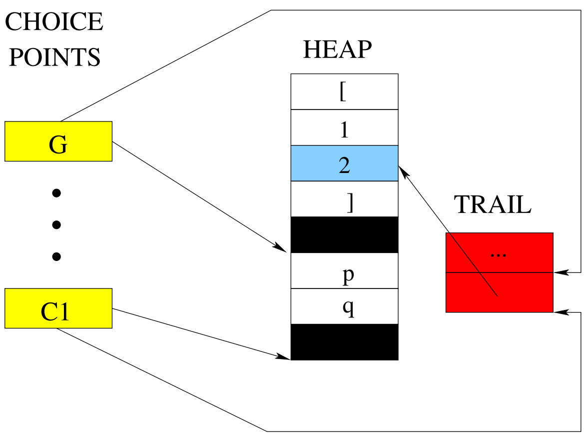
Suspension Based Implementation

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



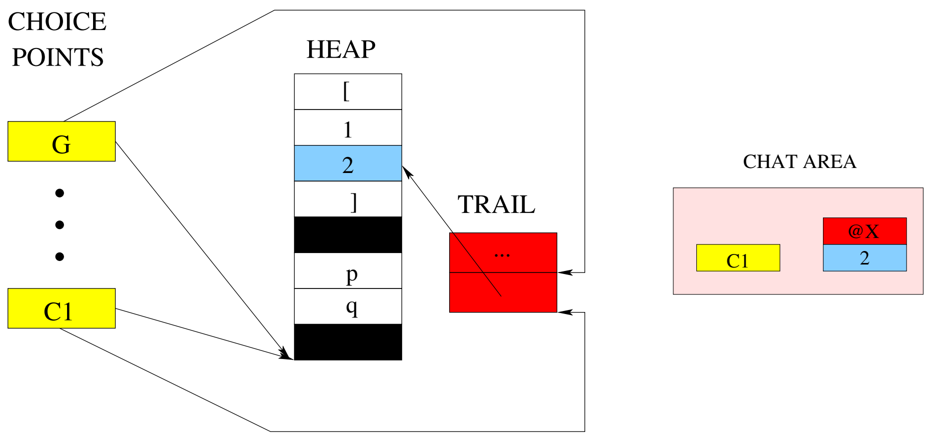
Suspension Based Implementation

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



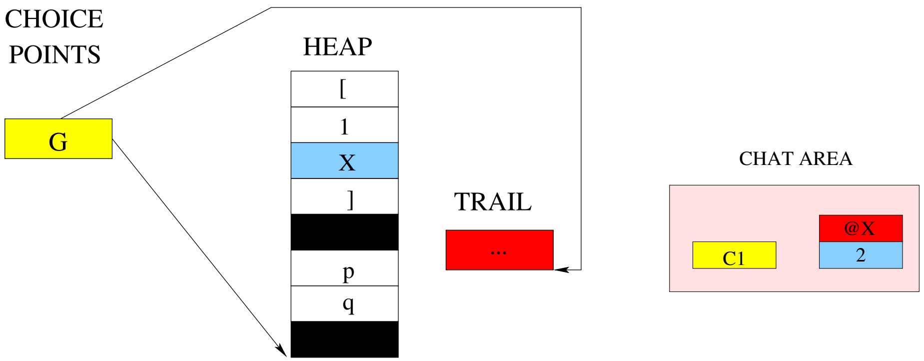
Suspension Based Implementation

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



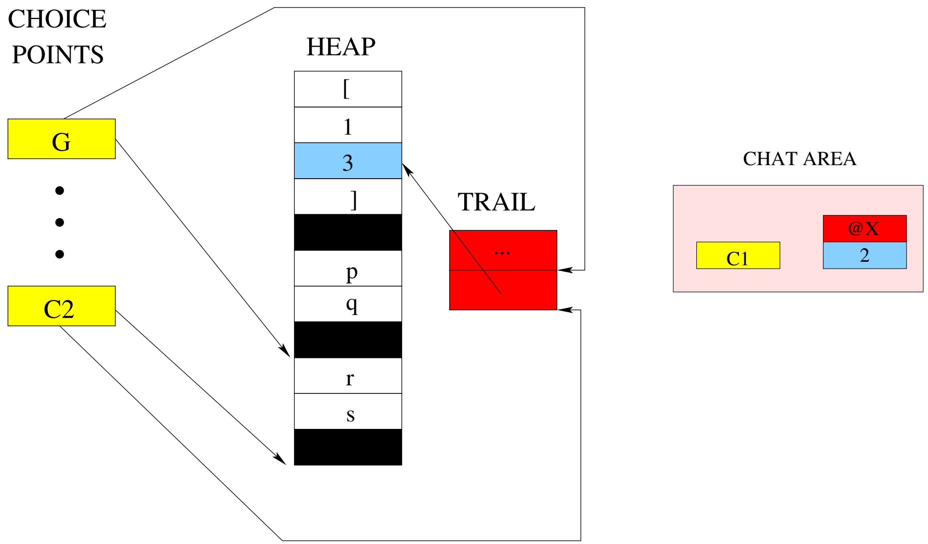
Suspension Based Implementation

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



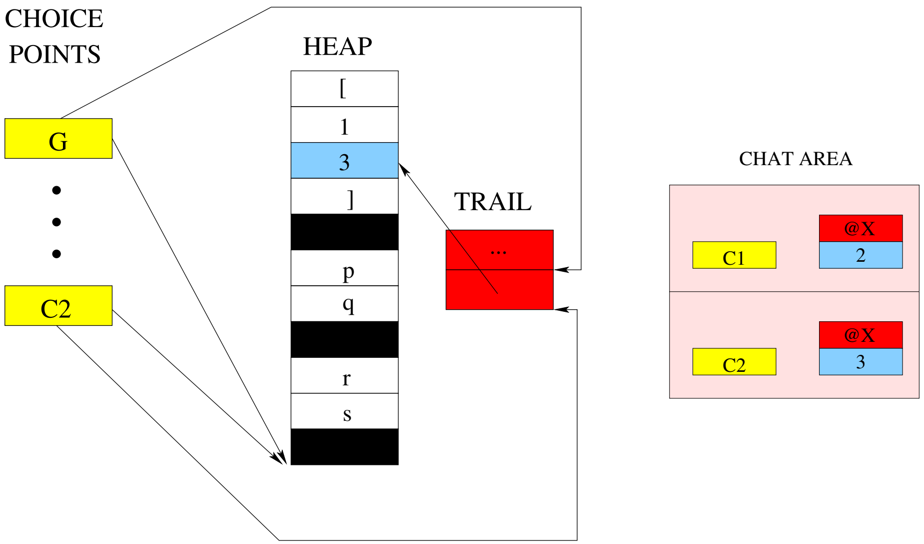
Suspension Based Implementation

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



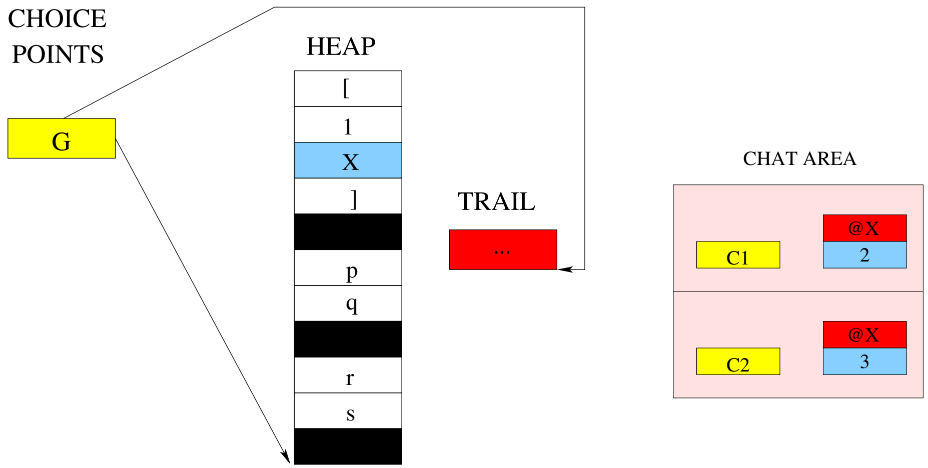
Suspension Based Implementation

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



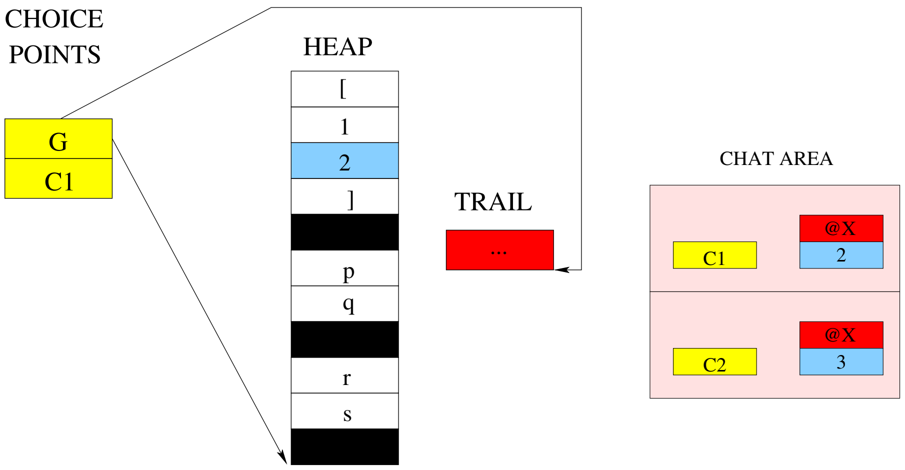
Suspension Based Implementation

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



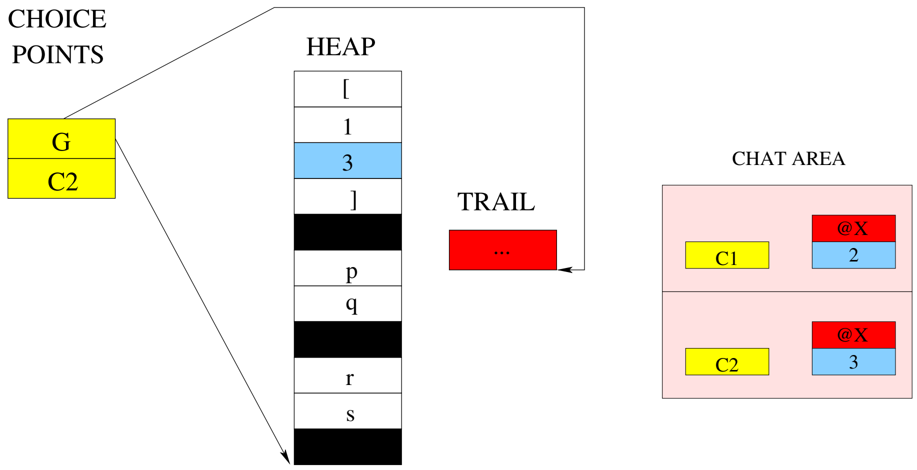
Suspension Based Implementation

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



Suspension Based Implementation

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



Open Research Topics

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.