Planning as Tabled Logic Programming

Neng-Fa Zhou, Roman Barták, Agostino Dovier
Logic Programming for Planning

- PLANNER [Hewitt69], “a language for proving theorems and manipulating models in a robot”
- Prolog for planning [Kowalski79, Warplan76]
  - Not so successful
- ASP-based planners [Lifschitz02]
- Action languages [PDDL, HTN, $A$, $K$, …]
- Tabled logic programming for individual cases
Goal and Outline

- **Goal**
  - Tabled logic programming for planning
  - Encodings for the IPC’14 domains

- **Outline**
  - A brief overview of Picat
  - Picat’s planner and its implementation
  - Modeling techniques and examples
  - Experimental results
  - Conclusion
Overview of Picat

- Why the name “PICAT”?
  - Pattern-matching, Intuitive, Constraints, Actors, Tabling

- Core logic programming concepts
  - Logic variables (arrays and maps are terms)
  - Implicit pattern-matching and explicit unification
  - Explicit non-determinism

- Language constructs for scripting and modeling
  - Functions, loops, and list comprehension

- Modules for combinatorial search
  - The \texttt{cp}, \texttt{sat}, and \texttt{mip} modules for CSPs
  - The \texttt{planner} module for planning
Data Types

Basic data types

A = {{1, _3d4, _3d8}, {_3e0, _3e4, 5}}

Picat> M = new_map([alpha=1, beta=2]), M.get(alpha) = A
M = (map)[alpha = 1,beta = 2]
A = 1
Logic Programming in Picat

member(X,[Y|_]) ?=\> X = Y.
member(X,[_|L]) => member(X,L).

append([],Ys,Zs) ?=\> Ys = Zs.
append([X|Xs],Ys,Zs) =>
Zs = [X|ZsR],
append(Xs,Ys,ZsR).

- Pattern-matching rules
- Explicit unification
- Explicit non-determinism
Functional programming in Picat

- Dynamically typed
- List comprehension
- Strict (not lazy)
- Higher-order functions

```
power_set([]) = [[]].
power_set([H|T]) = P1++P2 =>
  P1 = power_set(T),
  P2 = [[H|S] : S in P1].

qsort([]) = [].
qsort([H|T]) = qsort([E : E in T, E=<H])++
  [H]++
  qsort([E : E in T, E>H]).
```
Scripting in Picat

```
pascal =>
    print("enter an integer:"),
    N = read_int(),
    foreach(I in 0..N)
        Num := 1,
        foreach(K in 1..I+1)
            printf("%d ",Num),
            Num := Num*(I-K+1) div K
        end,
    end,
    nl
end.
```

- SSA (Static Single Assignment)
- Loops

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```
Constraint Programming in Picat

import cp.
import sat.
import mip.

send_more_money =>
  Vars = [S,E,N,D,M,O,R,Y],
  Vars :: 0..9,
  all_different(Vars),
  S #!= 0,
  M #!= 0,
  1000*S+100*E+10*N+D+1000*M+100*O+10*R+E
  #!= 10000*M+1000*O+100*N+10*E+Y,
  solve(Vars),
  % label variables
  writeln(Vars).

- Common interface to CP, SAT, and MIP

\[
\begin{array}{c}
\text{SEND} \\
+ \text{MORE} \\
\text{MONEY}
\end{array}
\begin{array}{c}
\text{9567} \\
+ \text{1085} \\
\text{10652}
\end{array}
\]
Dynamic Programming in Picat

- Linear tabling
- Mode-directed tabling
- Term sharing

```picat
table (fib, 1) = 1.
fib(2) = 1.
fib(N) = fib(N-1)+fib(N-2).

table (+,-,min)
path(S,Plan,Cost),final(S) =>
    Plan=[],Cost=0.
path(S,Plan,Cost) =>
    action(S,NextS,Action,ACost),
    path(NextS,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ACost.
```
Planning in Picat

Based on tabling
- Allows use of structures to represent states
- Supports domain knowledge and heuristics
- Provides search predicates
  - Depth-unbounded & depth-bounded
  - IDA & branch-and-bound

import planner.

go =>
    S0=[s,s,s,s],
    best_plan(S0,Plan),
    writeln(Plan).

final([n,n,n,n]) => true.

action([F,F,G,C],S1,Action,Cost) ?=>
    Action=farmer_wolf,
    Cost = 1,
    opposite(F,F1),
    S1=[F1,F1,G,C],
    not unsafe(S1).

...
The planner Module

- The planner module is based on tabling
- It provides a level of abstract that hides tabling
- Users only need to define `final/1` and `action/4`
  - `final(State)`
    - True if State is a final state
  - `action(State, NextState, Action, ActionCost)`
    - encodes the state space
The **planner** Module

- **Search predicates**
  - `plan_unbounded(State,Limit,Plan,PlanCost)`
    - Depth-unbounded search, return the first plan
  - `best_plan_unbounded(State,Limit,Plan,PlanCost)`
    - Depth-unbounded search, return the best plan
  - `plan(State,Limit,Plan,PlanCost)`
    - Resource-bounded search, return the first plan
  - `best_plan(State,Limit,Plan,PlanCost)`
    - Resource-bounded iterative-deepening search
  - `best_plan_bb(State,Limit,Plan,PlanCost)`
    - Resource-bounded branch-and-bound search
  - `current_resource()`
    - Return the current resource amount

Tabled Planning

ICLP’15
Depth-unbounded Search

table (+,-,min)
best_plan_unbounded(S,Plan,Cost),final(S) =>
   Plan=[],Cost=0.
best_plan_unbounded(S,Plan,Cost) =>
   action(S,NextS,Action,ACost),
   best_plan_unbounded(NextS,Plan1,Cost1),
   Plan = [Action|Plan1],
   Cost = Cost1+ACost.
Resource-Bounded Search

- Special treatment of the resource limit argument
  - It is tabled but not used in variant checking

$S^R$ is the current node, where $S$ is the state and $R$ is the resource limit. $S^{R'}$ failed before. $S^R$ can be failed immediately if $R \leq R'$. 
Modeling Techniques

- Find good (structured) representation for states
  - leaves out unimportant information
  - breaks symmetries
  - is good for precondition checking and state generation
  - facilitates term sharing
Modeling Techniques (Con.)

- Use control knowledge (reduce nondeterminism)
  - Deterministic actions
  - Dependent actions
    - Automaton-driven action sequences
  - Partial order reduction
- Use heuristics
  - A state should not be expanded if the travel from it to the final state costs more than the limit
## Solutions for IPC’14 Domains

(Artworks by Agostino Dovier)

<table>
<thead>
<tr>
<th>Barman</th>
<th>Citycar</th>
<th>Tetris</th>
<th>Parking</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Barman" /></td>
<td><img src="image" alt="Citycar" /></td>
<td><img src="image" alt="Tetris" /></td>
<td><img src="image" alt="Parking" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Childsnack</th>
<th>Floortile</th>
<th>Cavediving</th>
<th>Transport</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Childsnack" /></td>
<td><img src="image" alt="Floortile" /></td>
<td><img src="image" alt="Cavediving" /></td>
<td><img src="image" alt="Transport" /></td>
</tr>
</tbody>
</table>

**ICLP’15**

**Tabled Planning**
The Transport Domain

- Setting
  - A weighted graph
  - Initial and destination locations of packages
  - Initial locations and capacities of trucks
- Possible actions: load, unload, drive
- An optimal plan normally requires trucks to cooperate.
PDDL Encoding

(define (domain transport)
    
    (:action drive
        :parameters (?v - vehicle ?l1 ?l2 - location)
        :precondition (and
            (at ?v ?l1)
            (road ?l1 ?l2)
        )
        :effect (and
            (not (at ?v ?l1))
            (at ?v ?l2)
            (increase (total-cost) (road-length ?l1 ?l2))
        )
    )

    (:action pick-up
        ...
    )

    (:action drop
        ...
    )
)

A state is represented as a set of flat facts (fluents)

Impossible to describe determinism or heuristics

Rely on the planner to learn
  - Determinism
  - Symmetry
  - Heuristics

Used in IPC

Practical applications are rare
Picat Encoding

State representation

\{Trucks, Packages\}

- Trucks: [Truck1, Truck2, ...]
  - Trucki: [Loci, Destsi, Capi]
- Packages: [Package1, Package2, ...]
  - Packagei: (Loci, Desti)

- Truck’s id and Package’s id are removed.
- Loaded packages have the same location as the truck
- Trucks and Packages are sorted
Example

main =>
Trucks = [[c1,[],2],[c2,[],3]],
Packages = [(c1,c2),(c1,c2),(c2,c5),(c3,c1)],
best_plan({Trucks,Packages},Plan).
Picat Encoding

- **final/1**

  ```picat
  final({Trucks,[]}) =>
  foreach([_Loc,Dests,_Cap] in Trucks)
    Dests == []
  end.
  ```

- No waiting packages
- No loaded packages (trucks are all empty)
Picat Encoding

```
action/4

action({Trucks, Packages}, NextState, Action, ACost) ?=>
  Action = $load(Loc), ACost = 1,
  select([Loc, Dests, Cap], Trucks, TrucksR),
  length(Dests) < Cap,
  select((Loc, Dest), Packages, PackagesR),
  NewDest = insert_ordered(Dests, Dest),
  NewTrucks = insert_ordered(TrucksR, [Loc, NewDest, Cap]),
  NewPackages = insert_ordered(Packages, (Loc, Dest)),
  NextState = {NewTrucks, PackagesR},

action({Trucks, Packages}, NextState, Action, ACost) ?=>
  Action = $unload(Loc), ACost = 1,
  select([Loc, Dests, Cap], Trucks, TrucksR),
  select(Dest, Dests, DestsR),
  NewTrucks = insert_ordered(TrucksR, [Loc, DestsR, Cap]),
  NewPackages = insert_ordered(Packages, (Loc, Dest)),
  NextState = {NewTrucks, NewPackages},

action({Trucks, Packages}, NextState, Action, ACost) =>
  Action = $drive(Loc, NextLoc),
  select([Loc|Tail], Trucks, TrucksR),
  road(Loc, NextLoc, ACost),
  NewTrucks = insert_ordered(TrucksR, [NextLoc|Tail]),
  NextState = {NewTrucks, Packages}.
```
A package can be deterministically unloaded if its destination is the same as the truck's location.

\[
\text{action}([\text{Trucks},\text{Packages}],\text{NextState},\text{Action},\text{ActionCost}), \\
\text{select}([\text{Loc},\text{Dest},\text{Cap}],\text{Trucks},\text{TrucksR}), \\
\text{select}(\text{Loc},\text{Dest},\text{DestR}) \quad % \text{unload it deterministically} \\
\Rightarrow \\
\text{Action} = \$\text{unload}(\text{Loc}), \\
\text{ActionCost} = 1, \\
\text{NewTrucks} = \text{insert}_{\text{ordered}}(\text{TrucksR},[\text{Loc},\text{DestR},\text{Cap}]), \\
\text{NextState} = \{\text{NewTrucks},\text{Packages}\}.
\]
Heuristics

One admissible but conservative heuristic:

- The load and upload costs of all of the packages plus the maximum of the shortest distances of the packages’ locations to their destinations.
### Experimental Results

(http://picat-lang.org/ipc14/)

<table>
<thead>
<tr>
<th>Domain</th>
<th># insts</th>
<th>Picat</th>
<th>Picat-nt</th>
<th>Picat-nh</th>
<th>Symba</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barman</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>Cave</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>Childsnack</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>Citycar</td>
<td>20</td>
<td>20</td>
<td>17</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>Floortile</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>GED</td>
<td>20</td>
<td>20</td>
<td>19</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>Parking</td>
<td>20</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tetris</td>
<td>17</td>
<td>13</td>
<td>13</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Transport</td>
<td>20</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

The table shows the number of optimally solved instances for different domains. The results include the number of instances solved by Picat, Picat-nt, Picat-nh, and Symba. The table indicates that Picat and Picat-nh are the IPC 2014 winners, and none of the solvers used heuristics or tabling. 

---

**ICLP'15 Tabled Planning**

27
Conclusion

- Picat is powerful and efficient modeling language for planning
  - Solving techniques (tabling, term sharing, resource-bounded search)
  - Modeling techniques (structural representation, symmetry breaking, domain knowledge, and heuristics)

- Future work
  - New implementation and modeling techniques for coping with state explosion
  - Implement action languages in Picat