Scripting and Modeling with Picat

Neng-Fa Zhou
CUNY Brooklyn College and Graduate Center

joint work with Jonathan Fruhman and Hakan Kjellerstrand
Why Picat?

- Many complaints about Prolog
  - Implicit unification and non-determinism are difficult
  - Cuts and dynamic predicates are non-logical
  - Lack of constructs for programming everyday things

- No satisfactory successors
  - Prolog extensions are ad hoc (e.g., loops in B-Prolog)
  - Mercury requires too many declarations
  - Erlang abandons non-determinism in favor of concurrency
  - Oz’s syntax is strange and implicit laziness is difficult
  - Curry is too close to Haskell
Features of PICAT

- **Pattern-matching**
  - Predicates and functions are defined with pattern-matching rules

- **Imperative**
  - Assignments, loops, list comprehensions

- **Constraints**
  - CP, SAT and LP/MIP

- **Actors**
  - Action rules, event-driven programming, actor-based concurrency

- **Tabling**
  - Memoization, dynamic programming, planning, model-checking
Outline of Talk

- A brief overview of Picat
- Scripting with Picat
- Modeling with Picat
  - CSP modeling
  - Dynamic programming
  - Planning
- Conclusion
Data Types

- Variables – plain and attributed
  - $x1 \_\_ab$

- Primitive values
  - Integer and float
  - Atom
    - $x1 \_'\_ab' \'$%' '你好'

- Compound values
  - List
    - [17, 3, 1, 6, 40]
    - String “abc” is the same as [a, b, c]
  - Structure
    - $\text{triangle}(0.0, 13.5, 19.2)$
The Type Hierarchy

- **term**
  - **atomic**
    - **atom**
    - **number**
      - **integer**
      - **real**
  - **compound**
    - **list**
    - **string**
    - **struct**
      - **array**
      - **map**
    - **var**
      - **attr_var**
      - **dvar**
Creating Structures and Lists

- **Structure**
  Picat> P = new_struct(point, 3)
  P = point(_3b0, _3b4, _3b8)
  Picat> S = $student(marry, cs, 3.8)

- **List Comprehension**
  Picat> L = [(A, I) : A in [a, b], I in 1..2]
  L = [(a,1), (a,2), (b,1), (b,2)]

- **Range**
  Picat> L = 1..2..10
  L = [1, 3, 5, 7, 9]

- **String**
  Picat> write("hello ++"world")
  [h, e, l, l, o, ' ', w, o, r, l, d]

- **Array**
  Picat> A = new_array(2, 3)
  A = {{_3d0, _3d4, _3d8}, {_3e0, _3e4, _3e8}}

- **Map**
  Picat> M = new_map([alpha= 1, beta=2])
  M = (map)[alpha = 1, beta = 2]
Index Notation

\[ X[i_1, \ldots, i_n] : X \text{ references a compound value} \]

```
Picat> L = [a,b,c,d], X = L[2]
X = b

Picat> S = $student(marry,cs,3.8), GPA=S[3]
GPA = 3.8

Picat> A = {{1, 2, 3}, {4, 5, 6}}, B = A[2, 3]
B = 6
```
List Comprehension

\[ T : E_1 \text{ in } D_1, \text{Cond}_n, \ldots, E_n \text{ in } D_n, \text{Cond}_n \]

Picat> L = [X : X in 1..5].
L = [1,2,3,4,5]

Picat> L = [(A,I): A in [a,b], I in 1..2].
L = [(a,1),(a,2),(b,1),(b,2)]

Picat> L = [X : I in 1..5]  % X is local
L = [_bee8,_bef0,_bef8,_bf00,_bf08]

Picat> X=X, L = [X : I in 1..5]  % X is non-local
L = [X,X,X,X,X,X]
OOP Notation

Picat> Y = 13.to_binary_string()
Y = ['1', '1', '0', '1']

Picat> Y = 13.to_binary_string().reverse()
Y = ['1', '0', '1', '1']

% X becomes an attributed variable
Picat> X.put(age, 35), X.put(weight, 205), A = X.age
A = 35

%X is a map
Picat> X = new_map([age=35, weight=205]), X.put(gender, male)
X = (map)([age=35, weight=205, gender=male])

Picat> S = $point(1.0, 2.0)$, Name = S.name, Arity = S.length
Name = point
Arity = 2

Picat> I = math.pi % module qualifier
I = 3.14159
Predicates

- Backtracking (explicit non-determinism)

\[
\text{member}(X, [Y|\_]) \implies X=Y.
\text{member}(X, [\_|L]) \implies \text{member}(X,L).
\]

Picat> member(X, [1,2,3])
X = 1;
X = 2;
X = 3;
no

- Control backtracking

Picat> once(member(X, [1,2,3]))
Predicate Facts

- Facts must be ground
- A call with insufficiently instantiated arguments fails
  - Picat> edge(X,Y)
    no

index(+,-) (-,+)
edge(a,b).
edge(a,c).
edge(b,c).
edge(c,b).
edge(a,Y) ?=> Y=b.
edge(a,Y) => Y=c.
edge(b,Y) => Y=c.
edge(c,Y) => Y=b.
edge(X,b) ?=> X=a.
edge(X,c) ?=> X=a.
edge(X,c) => X=b.
edge(X,b) => X=c.
Functions

- Always succeed with a return value
- Non-backtrackable

\[
\begin{align*}
    \text{fib}(0) &= F \Rightarrow F=1. \\
    \text{fib}(1) &= F \Rightarrow F=1. \\
    \text{fib}(N) &= F, N>1 \Rightarrow F=\text{fib}(N-1)+\text{fib}(N-2).
\end{align*}
\]

- Function facts

\[
\begin{align*}
    \text{fib}(0) &= 1. \\
    \text{fib}(1) &= 1. \\
    \text{fib}(N) &= \text{fib}(N-1)+\text{fib}(N-2).
\end{align*}
\]
Assignments

- $X[I_1, \ldots, I_n] := \text{Exp}$
  Destructively update the component to $\text{Exp}$.
  Undo the update upon backtracking.

- $\text{Var} := \text{Exp}$
  The compiler changes it to $\text{Var}' = \text{Exp}$ and replace all subsequent occurrences of $\text{Var}$ in the body of the rule by $\text{Var}'$.

-test $\Rightarrow$ $X = 0$, $X := X + 1$, $X := X + 2$, write($X$).

-test $\Rightarrow$ $X = 0$, $X_1 = X + 1$, $X_2 = X_1 + 2$, write($X_2$).
Loops

- Types
  - `foreach(E1 in D1, ..., En in Dn) Goal end`
  - `while (Cond) Goal end`
  - `do Goal while (Cond)`

- Loops provide another way to write recurrences
- A loop forms a name scope: variables that do not occur before in the outer scope are local.
- Loops are compiled into tail-recursive predicates
Loops (Example) sum_even(L)

- Using a loop
  ```
  sum_even(L) = Sum =>
  S = 0,
  foreach (X in L)
    if even(X) then S := S + X end
  end,
  Sum = S.
  ```

- Using a list comprehension
  ```
  sum_even(L) = sum([X : X in L, even(X)]).
  ```
Tabling

- Predicates define relations where a set of facts is implicitly generated by the rules.
- The process of fact generation might never end, and can contain a lot of redundancy.
- Tabling memorizes calls and their answers in order to prevent infinite loops and to limit redundancy.
Tabling (examples)

table
fib(0)=1.
fib(1)=1.
fib(N)=fib(N-1)+fib(N-2).

table(+,+,−,min)
shortest_path(X,Y,Path,W) ?=>
Path = [(X,Y)],
edge(X,Y,W),
shortest_path(X,Y,Path,W) =>
Path = [(X,Z)|PathR],
edge(X,Z,W1),
shortest_path(Z,Y,PathR,W2),
W = W1+W2.
Modules

module M.
import M1,M2,...,Mn.

- The declared module name and the file name must be the same
- Files that do not begin with a module declaration are in the global module
- Atoms and structure names are global
- Picat has a global symbol table for atoms, a global symbol table for structure names, and a global symbol table for modules
- Each module has its own symbol table for the public predicates and functions
Supported Modules

- Pre-loaded and pre-imported
  - basic, math, io, sys

- Pre-loaded but names are not pre-imported
  - cp, planner, sat, os, util

- Not pre-loaded or pre-imported
  - Setting of PICATPATH is needed
Status of the Implementation

- Based on the B-Prolog engine
  - Over 20+ years of R/D
- System size
  - 55,000 LOC in C
  - 45,000 LOC in Picat
- Other system features
  - Debugger
  - Garbage collector
  - Big integers
Outline of Talk

- A brief overview of Picat
- **Scripting with Picat**
- Modeling with Picat
  - CSP modeling
  - Dynamic programming
  - Planning
- Conclusion
Traverse a Directory Tree

```python
import os.

traverse(Dir), directory(Dir) =>
    List = listdir(Dir),
    printf("Inside %s\n",Dir),
    foreach (File in List)
        printf("    %s\n",File)
    end,
    foreach (File in List, File != ".", File != "..")
        FullName = Dir ++ [separator()] ++ File,
        traverse(FullName)
    end.
    traverse(_Dir) => true.
```
Input Rows of Integers into an Array

import util.

input_data(Tri) =>
    Lines = read_file_lines("triangle.txt");
    Tri = new_array(Lines.length),
    I = 1,
    foreach(Line in Lines)
        Tri[I] = Line.split().map(to_integer).to_array(),
        I ::= I+1
    end.

foreach({I,Line} in zip(1..Lines.length,Lines))
    Tri[I] = Line.split().map(to_integer).to_array(),
end.
Outline of Talk

- A brief overview of Picat
- Scripting with Picat
- Modeling with Picat
  - Dynamic programming
  - Planning
  - CSP modeling
- Conclusion
The Omelet Problem
(Or The N-Eggs Problem)

```
table (+,+,min)
    omelet(_, 0, NTry) => NTry = 0.
    omelet(_, 1, NTry) => NTry = 1.
    omelet(1, H, NTry) => NTry = H.
    omelet(N, H, NTry) =>
        between(1, H, L), % make a choice
        omelet(N-1, L-1, NTry1), % the egg breaks
        omelet(N, H-L, NTry2), % the egg survives
        NTry is max(NTry1, NTry2) + 1.
```

http://www.datagenetics.com/blog/july2012/
Maximum Path Sum
(Euler Project 18 and 67)
	able (+,+,max,nt)
path(Row,Col,Sum,Tri), Row==Tri.length =>
    Sum=Tri[Row,Col].
path(Row,Col,Sum,Tri) ?=>
    path(Row+1,Col,Sum1,Tri),
    Sum = Sum1+Tri[Row,Col].
path(Row,Col,Sum,Tri) =>
    path(Row+1,Col+1,Sum1,Tri),
    Sum = Sum1+Tri[Row,Col].

9/23/2013 (IBM PL DAY)  27
The **planner** Module

- Useful for solving planning problems
  - \( \text{plan}(\text{State}, \text{Limit}, \text{Plan}, \text{PlanCost}) \)
  - \( \text{best_plan}(\text{State}, \text{Limit}, \text{Plan}, \text{PlanCost}) \)
  - ... 
- Users only need to define **final**/1 and **action**/4
  - \( \text{final}(\text{State}) \) is true if State is a final state
  - \( \text{action}(\text{State}, \text{NextState}, \text{Action}, \text{ActionCost}) \) encodes the state transition diagram
- Uses tabling with the early termination and resource-bounded search techniques to speedup search
Ex: The Farmer’s Problem

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

...
Tabling is More Effective Than SAT for Planning?

- **Nomystery** ([picat-lang.org/asp/nomystery.pi](http://picat-lang.org/asp/nomystery.pi))
  - Picat solves all of the 30 instances
  - Clasp solves only 17 of the 30 instances
  - On solved instances, Picat is more than 100 times faster than Clasp

- **Sokoban** ([picat-lang.org/asp/sokoban.pi](http://picat-lang.org/asp/sokoban.pi))
  - Picat solves all of the 30 instances
  - Clasp solves only 14 of the 30 instances

- **Ricochet Robots** ([picat-lang.org/asp/ricochet.pi](http://picat-lang.org/asp/ricochet.pi))
  - Both Picat and Clasp solve all of the 30 instances
  - Picat is several times faster than Clasp despite that its encoding is much simpler
Constraints

- Picat can be used for constraint satisfaction and optimization problems

- Constraint Problems
  - Generate variables
  - Generate constraints over the variables
  - Solve the problem, finding an assignment of values to the variables that matches all the constraints

- Picat can be used as a modeling language for CP, SAT, LP/MIP
  - Loops are helpful for modeling
SEND + MORE = MONEY

import cp.

go =>

Vars=[S,E,N,D,M,O,R,Y], % generate variables
Vars in 0..9, % define the domains
all_different(Vars), % generate constraints
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), % search
writeln(Vars).
import cp.

queens3(N, Q) =>
    Q = new_list(N),
    Q in 1..N,
    all_different(Q),
    all_different([Q[I]-I : I in 1..N]),
    all_different([Q[I]+I : I in 1..N]),
    solve([ff], Q).
The Number Link Problem

(picat-lang.org/asp/numberlink_b.pi)

Solved with the sat module of Picat and the Lingeling solver
Conclusion

- Picat is a logic-based multi-paradigm language
- Picat can be used as a scripting language
  - Future work: add modules for programming Web services
- Picat can also be used as a modeling language for DP, planning, and CSP
  - Future work: add modules for other solvers
## Picat Vs. Haskell

<table>
<thead>
<tr>
<th>Commonalities</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern-matching</td>
<td>untyped vs. typed</td>
</tr>
<tr>
<td>strings, list comprehensions</td>
<td>strict vs. lazy</td>
</tr>
<tr>
<td>tail-recursion optimization</td>
<td>assignments vs. monads</td>
</tr>
<tr>
<td>Picat supports FP</td>
<td>multi-paradigm vs. pure FP</td>
</tr>
</tbody>
</table>

- **Picat is more suitable to symbolic computations**
  - Explicit unification
  - Explicit non-determinism
  - Tabling
  - Constraints
Picat Vs. Prolog

- Picat is more expressive
  - Functions, arrays, maps, loops, and list comprehensions
- Picat is more scalable because pattern-matching facilitates indexing rules
- Picat is arguably more reliable than Prolog
  - Explicit unification and non-determinism
  - Functions don’t fail (at least built-in functions)
  - No cuts or dynamic predicates
  - No operator overloading
  - A simple static module system
Resources

- Users’ Guide

- Hakan Kjellerstrand’s Picat Page
  - http://www.hakank.org/picat/

- Examples
  - http://picat-lang.org/download/exs.pi

- Modules

- Projects