Picat: A Logic-based Multi-paradigm Language

Håkan Kjellerstrand
Independent Researcher, Malmoe, Sweden
E-mail: hakank@gmail.com
June, 2014

Abstract

Picat is a new and interesting programming language, combing many different and exciting programming paradigms: logic programming, imperative programming, constraint programming, functional programming, tabling, and planning. This paper gives a personal discussion of some of Picat's features with several code examples. Keywords: Picat, Constraint Programming, Logic Programming, Planning, Prolog.

1 INTRODUCTION

Picat [1] is a new programming language created by Neng-Fa Zhou and Jonathan Fruhman. The specification was published in December 2012 in [3]. The first alpha version was made available in May 2013 and the first stable version 0.l was released on 2014-02-01. The current version is 0.3, which was released on 2014-05-23.

I started to test Picat in May 2013 when the first alpha version was available. As first, I tested Picat's support of Constraint Programming (CP), which is a programming paradigm that I am very interested in (see for example my Constraint Programming Blog [6]). Then, I quickly got fascinated by the complete system. In this paper, I will explain why.

The name Picat is an acronym and is explained in the quote below, taken from the Picat's web site [1]: *Picat is a general-purpose language that incorporates features from logic programming, functional programming, and scripting languages. The letters in the name summarize Picat's features*: *P*attern-matching, *I*mperative, *C*onstraints, *A*ctors, and *T*abling. I would also like to emphasize the following important paradigms/features: *Logic-based*, *Planning*, and *Functions*.

All the programs and models given in this paper, including hundreds of others, are available on my My Picat Page [2] and also in my GitHub repository: `https://github.com/hakank/hakank/tree/master/picat/`.

2 CONSTRAINTS

Picat has a high-level support for Constraint Programming, much like most CLP systems. Let's see how it looks like with some examples.

2.1 SEND+MORE=MONEY

Let's begin with the SEND+MORE=MONEY puzzle. In Picat it can be stated as follows:

```
import cp.

go =>
   sendmore(Digits),
   println(Digits).

sendmore(Digits) =>
   Digits = [S,E,N,D,M,O,R,Y],
   Digits :: 0..9,
   all_different(Digits),
   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(Digits).
```

Except for the `=>`, it's much like any C(L)P model:

- Domains of variables are declared by using the domain constraint $Variable$ `::` $Domain$.
- A special character (here `#`) is used to denote arithmetic constraints (e.g., `#>` denotes greater than).
- The `solve` predicate is used to label variables with values (labeling options can be given in `solve/2`).
- Global constraints are used whenever appropriate (`all_different/1`).

2.2 N-queens

Below is a model for the N-queens problem, a common and mandatory CP example:

```
import cp.
queens3(N, Q) =>
  Q=new_list(N),
  Q :: 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 argument `[$Q[I]-I : I in 1..N]` is a list comprehension. Note that here we must use `$Q[I]-I` (i.e. `Q[I]` preceded with a dollar character) since it is a term, not a function call. This special notation is needed because Picat also support functions.

Here are the CPU times for finding the first solutions for different values of N:

```
N=8: 0.0s, 23 backtracks
N=100: 0.03s, 22 backtracks
N=400: 0.44s, 10 backtracks
N=1000: 2.7s, 2 backtracks
N=1500: 8.7s 2889 backtracks
```

Picat is perhaps not the fastest CP system in the world, but it's quite fast.

2.3 Reifications and Indexing

Reification amounts to reasoning about the satisfibility of constraints. It is a very important concept in Constraint Programming. A nice syntax for stating reifications tend to considerably simplify modeling.

Below is a decomposition of the `alldifferent_except_0`(Xs) constraint, which is one of my favorite global constraints. This constraint is true if all the elements in Xs that are not equal to 0 (zero) are distinct.

```
alldifferent_except_0(Xs) =>
  foreach(I in 1..Xs.length-1, J in I+1..Xs.length)
    (Xs[I] #!= 0 #/\ Xs[J] #!= 0) #=> (Xs[I] #!= Xs[J])
  end.

go =>
  N = 5,  X = new_list(N),  X :: 0..N,
  alldifferent_except_0(X),
  solve(X),
  println(X).
```

The `#=>` operator denotes implication. This example also illustrates the use of indexing of lists. The index notation `Xs[I]` requires the index expression `I` to given an integer. If an index is a domain variable, then the constraint `element/3` should be used. Note that lists in Picat are singly linked lists, and `Xs[I]` takes `O(I)` time. For large collections, arrays should be used instead of lists. Also note that the `foreach` loop iterates over all the values in the range and there is no way to break out the loop. If it is required to iterate over values in a range until a condition is met, then the `while` loop or recursion should be used instead.

2.4 The Least Diff Problem, Optimization

The Least Diff problem is a simple optimization problem: Minimize the difference between ABCDE-FGHIJ, where the letters A..J are distinct digits.

```
import cp.
least_diff(L,Diff) =>
  L = [A,B,C,D,E,F,G,H,I,J],
  L :: 0..9,
```

```
all_different(L),
X #= 10000*A + 1000*B + 100*C + 10*D + E,
Y #= 10000*F + 1000*G + 100*H + 10*I + J,
Diff #= X - Y,
Diff #> 0,    % break symmetry
solve([$min(Diff)], L),
println(L).
```

The labeling option `$min(Diff)` (note the existence of the dollar character) gives the objective variable to be minimized.

The program prints an optimal solution. When developing more advanced models, it is sometimes useful to use the labeling option `$report(Goal)` to show how an optimal value is reached. Here is an example from the model `einav_puzzle.pi`:

```
solve($[min(TotalSum), report(printf("Found %d\n", TotalSum)), ffd],Vars),
```

2.5 Built-in Global Constraints

Picat supports the most common global constraints as shown in the list below. See Picat Guide [7], section 11.5 for the details.

`all_different(`*List*`)`
`all_distinct(`*List*`)`
`assignment(`*List*`)`
`circuit(`*List*`)`
`cond(`*Condition*`, `*ThenExp*`, `*ElseExp*`) #= `*Exp*
`count(`*Value*`,`*List*`,`*Rel*`,`*N*`)`
`cumulative(`*Starts*`,`*Durations*`,`*Resources*`, `*Limit*`)`
`diffn(`*RectangleList*`)`
`disjunctive_tasks(`*Tasks*`)`
`element(`*I*`,`*List*`, `*V*`)`
`global_cardinality(`*List*`, `*Pairs*`)`
`max(`*List*`) #= `*Exp*
`min(`*List*`) #= `*Exp*
`neqs(`*NegList*`)`
`serialized(`*Starts*`,`*Durations*`)`
`subcircuit(`*List*`)`
`sum(`*List*`) #= `*Exp*
`table_in(`*Vars*`, `*Relation*`)`
`table_notin(`*Vars*`, `*Relation*`)`

2.6 Labeling

Picat supports most of the standard variable and value labeling strategies, including `rand`, `rand_var`, and `rand_val`, which were introduced in version 0.3. For example, in `solve([ff,down], Vars)`, the option `ff` means *first-fail*, and `down` means to choose a value from the largest value to the smallest for a selected variable.

3 TABLING

Tabling here refers to memoization ("caching the results", see [5]). A simple example of tabling is the Fibonacci program.

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

The keyword `table` in the beginning means that the function is tabled. Without tabling this definition is very slow for larger N because of the exponential number of calls. When tabled, all the calls and their answers will be cached. In this way, subsequent calls are then resolved by simple table lookups, which can considerably speed up the execution. The following query calculates the 10000th Fibonacci number.

```
Picat> cl(fib)
Picat> time(F=fib(10000))
```

This call takes about 0.33s. Without tabling, the query would take "forever".

Note that in some cases using tabling may actually slow down programs because of the memory overhead involved, so one has to use tabling with caution and test both versions with and without tabling.

Another use of tabling is for dynamic programming. Here is an example for calculating the edit distance (taken from Picat's example `exs.pi`) where the third argument (D) is to be minimized for each pair of input arguments with the table mode `+`.

```
table(+,+,min)
edit([],[],D) => D=0.
edit([X|Xs],[X|Ys],D) =>  % copy
  edit(Xs,Ys,D).
edit(Xs,[_Y|Ys],D)   ?=>  % insert
  edit(Xs,Ys,D1),
  D=D1+1.
edit([_X|Xs],Ys,D) =>     % delete
  edit(Xs,Ys,D1),
  D=D1+1.
```

4 IMPERATIVE PROGRAMMING

Picat supports loops, which can be seen as a trademark in imperative programming. According to [3], the loop construct was one of the main reasons why Neng-Fa Zhou wanted to create a new programming language. Zhou was not satisfied with the foreach loop he introduced into B-Prolog [4]. One important feature that distinguishes Picat's foreach loop from B-Prolog's (as well as ECLiPSe's and SICStus Prolog's) is that the user does not have to explicitly declare either local or global variables. Scopes of variables are determined by the following rule: *Variables that occur only in a loop, but do not occur before the loop in the outer scope, are local to each iteration of the loop.* This rule makes loops much easier to use and read.

Here is an example for the Project Euler problem `#2` [8], where the task is to calculate the sum of all even Fibonacci numbers below 4000000:

```
euler2b =>
  I = 1,
  Sum = I,
  F = fib(I),
  while (F < 4000000)
     if F mod 2 == 0 then
        Sum := Sum + F
     end,
     I := I + 1,
     F := fib(I)
  end,
  writeln(Sum).
```

This program calculates the solution in 0.015s. The possibility to switch among different paradigms is a cool thing of Picat.

5 PATTERN MATCHING

Pattern matching in Picat is more influenced by pattern matching found in functional programming (e.g. Haskell) than by Prolog's pattern matching/unification. This might confuse some programmers with a Prolog background.

Here is a definition of quick-sort, which shows some of the pattern matching constructs in Picat.

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

The standard `take` function in functional programming can be defined as follows:

```
take(_Xs,0) = [].
take([],_N) = [].
take([X|Xs],N) = [X|take(Xs,N-1)].
```

6 NONDETERMINISM AND PROLOG INFLUENCES

Some of the above examples already show the clear influences from Prolog. Here we give more examples.

Picat, like Prolog, supports nondeterminism through a backtracking mechanism. This is a really great feature of Picat and is one of the reasons that I like Picat. Predicates using backtracking must be defined with `?=>` instead of `=>`.

6.1 Member

The following defines a nondeterministic predicate, `member2`. This predicate can be used both to check for membership and to retrieve elements from a given list.

```
member2(X,[Y|_]) ?=> X=Y.
member2(X,[_|L]) => member2(X,L)
```

Both this predicate and the built-in `member/2` predicate work as in Prolog.

```
Picat> member(2, [1,2,3])
yes
Picat> member(4, [1,2,3])
no
Picat> member(X, [1,2,3])
  X=1;
  X=2;
  X=3;
  no
```

The next solution will be shown when the user enters a ; (semicolon) after a solution in the Picat shell.

One difference between Picat's pattern matching and Prolog's is that no variables in calls can get bound during Picat's pattern matching, and such bindings must be explicitly done in the body of a rule. For example, the definition of append/3 cannot be stated in Picat as nicely as in Prolog.

```
append2(Xs,Ys,Zs) ?=> Xs=[], Ys=Zs.
append2(Xs,Ys,Zs) => Xs=[X|XsR], Zs=[X|ZsR], append2(XsR,Ys,ZsR).
```

This version has the same non-deterministic behavior as in Prolog, but it does not deterministically branches on the first argument in case it is non-variable. The implementation of the built-in `append` takes care of both the input and output modes of the first argument.

6.2 Predicate Facts

Predicate facts are bodyless definitions akin to Prolog's "data definition". Predicate facts must are prepended by an index declaration to make them accessible as data (this explicit declaration is one of the differences from Prolog). For example, here is a stripped down version of the transitive closure example `transitive_closure.pi`:

```
top ?=>
  reach(X,Y),
  writeln([X,Y]),
  fail.
top => true.

table
reach(X,Y) ?=> edge(X,Y).
reach(X,Y)  => edge(X,Z),reach(Z,Y).

index (-,-) (+,-)
edge(1,2).   edge(1,3).   edge(2,4).  edge(2,5).  edge(3,6).
```

The index declaration gives the modes to the compiler for converting the facts into pattern-matching rules. The mode (-) indicates a variable and the mode (+) indicates a ground term. As mentioned above, the use of `table` will cache the `reach` predicate. Using `table` also avoids looping in case the graph contains cycles.

6.3 Other Nondeterministic Predicates

Here are the nondeterministic built-in predicates in Picat. Many of these are wrappers of the underlying B-Prolog predicates.

- `append`(W, X, Y, Z)
- `append`(X, Y, Z)
- `between`$(From, To, X)$
- `find`$(String, Substring, From, To)$
- `find_ignore_case`$(String, Substring, From, To)$
- `indomain`(Var) (CP)
- `indomain_down`(Var) (CP)
- `member`$(Term, List)$
- `nth`$(I, List, Val)$
- `permutation`$(List, Perm)$
- `repeat`
- `select`$(X, List, ResList)$
- `solve`$(Options, Vars)$ (CP)
- `solve`$(Vars)$ (CP)
- `statistics`$(Name, Value)$

Picat also has a findall function: $List$ = `findall`$(X, predicate(X))$. Note that the Picat documentation generally recommends writing a function rather than a predicate in case only one answer is required since functions are generally easier to debug than predicates.

7 COMMON INTERFACE TO CP, SAT AND MIP SOLVERS

Picat also supports solving constraint problems using SAT through an interface with Lingeling and MiniSat, and using MIP through an interface with GLPK. In order to switch to a different solver, we just need to import the solver module. The CP models given above also work if `import cp` is changed to `import sat` or `import mip`.

Note that when the `mip` solver is used all nonlinear constraints on integer-domain variables, including reification and global constraints, are linearized. A CP model that includes nonlinear constraints may perform poorly when the `mip` module is used. On the other hand, the SAT solver can be quite fast in certain cases. The `mip` module also supports constraints over real variables, which is not the case for `cp` or `sat`.

8 PLANNING

Another thing of Picat that I really enjoy is the planner module for modeling and solving planning problems. Here are the steps we need to follow in order to model a planning problem:

- Import the module: `import planner`.
- Define the `action` predicate for the legal moves: `action`$(FromState, ToState, Move, Cost)$.
- Define the `final` predicate for goal state(s): `final(`$State$`)` succeeds if $State$ is a final state.
- Call one of the `plan` predicates with an initial state and other appropriate arguments.

The following gives a simple planning example (`test_planner_M12.pi`):

```
import planner.
go =>
  Init = [8,11,6,1,10,9,4,3,12,7,2,5],
  time(best_plan(Init, Plan)),
  writeln(Plan),
  writeln(len=Plan.length),
  nl.

final(Goal) =>
  Goal=[1,2,3,4,5,6,7,8,9,10,11,12].

% merge move
action([M1,M12,M2,M11,M3,M10,M4,M9,M5,M8,M6,M7],To,M,Cost) ?=>
  Cost=1, M=m, To=[M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12].
% reverse move
action([M12,M11,M10,M9,M8,M7,M6,M5,M4,M3,M2,M1],To,M,Cost) =>
  Cost=1,M=r,To=[M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12].
```

There are two actions: merge (perfect shuffle) and reverse. It is important to make the first rule of `action` backtrackable with `?=>`, because otherwise the second rule would never be used.

Picat finds the shortest solution for this problem instance (27 steps) in about 1s:

```
[m,m,m,r,m,m,m,r,m,m,m,m,r,m,r,m,r,m,m,r,m,m,r,m,m,m,r]
```

For this kind of "straight" planning problems, it's often much easier to model and much faster to solve using the planner module than using CP/SAT solvers.

Based on the planner module, I have written a GPS/STRIPS-inspired module that has support for add-lists and delete-lists. See my planning programs for some examples: `http://www.hakank.org/picat/#planning`.

9 OTHER FEATURES OF PICAT

Picat has many other features that are not covered in this paper. See the Picat User's Guide for a detailed description.

- Maps (i.e., hash tables), including global maps.
- Debugging and tracing, akin to Prolog's debug facilities.
- The `io` and `os` modules for handling files and directories. Reading files is often much easier in Picat than in standard Prolog
- An interface to PRISM (probabilistic reasoning and learning).
- The `math` module for standard math functions, including `primes(`Int`)`, `prime(`Int`)`, `random()`, `random2()`.
- The Picat shell, which is much like a Prolog shell

10 SO, WHY DO I LIKE PICAT?

To summarize, here are the reasons that make me like Picat: Support of different paradigms in one system; support for CP at a high level; the same encoding for CP, SAT and MIP; nondeterminism as seen in logic programming; imperative programming; functional programming (kind of); and the planner module. I enjoy programming in Picat because it suites my mindset very well.

REFERENCES

[1] http://picat-lang.org/

[2] http://www.hakank.org/picat/

[3] Neng-Fa Zhou: comp.lang.prolog, thread, "Picat: A New Logic-Based Language".

[4] http://www.probp.com/

[5] http://en.wikipedia.org/wiki/Memoization

[6] http://www.hakank.org/constraint_programmming_blog/

[7] http://picat-lang.org/download/picat_guide.pdf

[8] http://projecteuler.net