Canonicalizing High-Level Constructs in Picat

Neng-Fa Zhou[1(✉)] and Jonathan Fruhman[2]

[1] CUNY Brooklyn College and Graduate Center, New York, USA
zhou@sci.brooklyn.cuny.edu
[2] New York, USA

Abstract. Picat is a logic-based multi-paradigm dynamic language that integrates logic programming, functional programming, constraint programming, and scripting. The Picat language is underpinned by the core logic programming concepts, including logic variables, unification, and nondeterminism. Picat takes many constructs from other languages, among which functions, list and array comprehensions, loops, and assignments are convenient for scripting and modeling. This paper gives an overview of the language features of Picat, and shows how different language constructs are compiled into a canonical form.

1 Introduction

Picat is a simple, and yet powerful, logic-based multi-paradigm dynamic language. Picat was designed with the goal of creating a logic-based general-purpose programming language that overcomes the weaknesses of Prolog, is as powerful as Python and Ruby for scripting, and is on a par with OPL [6] and MiniZinc [10] for modeling combinatorial problems.

Like Prolog, Picat is based on the core logic programming concepts, including logic variables, unification, and nondeterminism realized through depth-first backtracking search. Picat departs from Prolog in many aspects. Picat uses pattern-matching rather than unification in the selection of rules. Unification might be a natural choice in Horn clause resolution for theorem proving [8], but its power is rarely needed for general programming tasks. In Picat, pattern-matching rules are fully indexed, while most Prolog implementations only index clauses on one argument; therefore, Picat can be more scalable than Prolog. Unification can be considered as an equation over terms [1], and just like constraints over finite domains, Picat supports unification as an explicit call.

Non-determinism, a powerful feature of logic programming, makes concise solutions possible for many problems, including the simulation of non-deterministic automata, the parsing of ambiguous grammars, and search problems. In Prolog, Horn clauses are backtrackable by default. As it is generally undecidable to detect determinism, programmers tend to excessively use the cut operator to prune unnecessary clauses. Picat supports explicit non-determinism, which renders the cut operator unnecessary. In Picat, rules are deterministic, unless they are explicitly annotated as backtrackable.

Picat supports functions, like many other logic-based languages, such as Curry [5] and Ciao [7]. In Prolog, a predicate defines a relation, and may succeed multiple times. It is common for queries to fail in Prolog without the system providing any clue about the source of the failure. Functions should be used instead of relations, unless multiple answers are required. It is more convenient to use functions instead of predicates, because (1) functions are guaranteed to succeed with a return value; (2) function calls can be nested; and (3) the directionality of functions often enhances the readability.

Picat provides arrays and loops, which are probably the features that are most unlike those of Prolog. In Prolog, in order to describe repetitions, programmers mainly rely on recursion, and occasionally rely on failure-driven loops and higher-order extensions [15]. The lack of powerful loop constructs has arguably made Prolog less acceptable to programmers than other languages. The extension of Prolog to support constraints has further revealed the weakness of Prolog as a modeling language. Early attempts to introduce arrays and loops into Prolog for modeling failed to produce a satisfactory language: most noticeably, array accesses are only treated as functions in certain contexts, and loops require the declaration of global variables in ECLiPSe [11] and local variables in B-Prolog [17].

Picat allows list comprehensions to be included as special functions in expressions in order to declaratively construct lists. Picat also supports the assignment operator `:=`, whose original motive was to facilitate the compilation of list comprehensions. A list comprehension is easily translated into a `foreach` loop in which an assignment is utilized to accumulate the constructed list. The decision to make the assignment operator available to programmers is controversial but pragmatic. The assignment operator in Picat has earned fondness among programmers for its simple semantics and convenience.

All of the language constructs, including functions, loops, comprehensions, and assignments, are provided as syntactic sugar in Picat. They are compiled away at compile time. This paper gives an overview of the language constructs of Picat, and shows how they are compiled into canonical-form pattern-matching rules.

2   The Picat Language

The name "Picat" is an acronym, and the letters in the name summarize Picat's features: **'P'** for pattern-matching, **'I'** for intuitive programming, **'C'** for constraints, **'A'** for action rules, and **'T'** for tabling. This section gives an overview of the language constructs of Picat. Some of Picat's features, such as action rules, tabling, and the tabling-based planner [18], are orthogonal to the language constructs, and will not be covered in this article. More details of the Picat language can be found in [20].

2.1   Data Types

Picat is a logic-based multi-paradigm programming language for general-purpose applications. Picat's core is underpinned by logic programming concepts, as seen

in Prolog, including *logic variables*, *unification*, and *backtracking*. Logic variables, like variables in mathematics, are value holders. A logic variable can be bound to any term, including another logic variable. Figure 1 gives the types of terms in Picat. Picat is a dynamically-typed language, which means that type checking occurs at runtime.
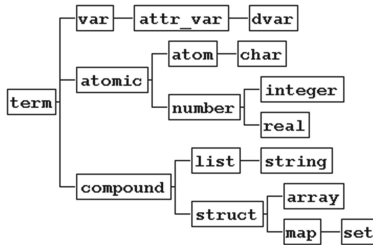

Fig. 1. Picat's data types

A variable name is an identifier that begins with a capital letter or the underscore; for example, X1 and _abc are variable names. The underscore itself _ is used for *anonymous variables*, and each occurrence of the underscore indicates a different variable.

An *atomic* value can be an *atom* or a *number*. An *atom* is a constant symbol. An atom name can be either unquoted or quoted. An unquoted name is an identifier that begins with a lower-case letter, followed by an optional string of letters, digits, and underscores. A quoted atom is a single-quoted sequence of arbitrary characters. For example, x1, x_1, '_abc', and 'a+b' are atom names. A number can be an *integer* or a *real number*. Picat supports big integers.

A *compound* value can be a *list* or a *structure*; for example, [a,b,c] is a list, and f(a,b,c) is a structure.[1] Lists are singly-linked lists. A *string* is a list of characters; for example, "a+b" is the same as [a,'+',b]. An *array* is a special structure; for example, {a,b,c} is an array. A *map* is a special structure that contains a set of key-value pairs, and a *set* is a special map that only contains keys; both are hash tables.

Each type provides a set of built-in functions and predicates. Each of the type names, except term and set, is a type-checking predicate. For example, list(L) tests if L is a list. Let L be a compound term. The index notation $L[I]$ is a special function that returns the Ith component of list L, with $L[1]$ referring to the first element of L. An index notation can take multiple subscripts. The *cons* operator [$H|T$] builds a new list by adding H to the front of T. The *concatenation* operator $L_1 ++ L_2$ returns the concatenated list of L_1 and L_2.

---

[1] A structure requires a preceding dollar symbol, as in $f(a,b,c), to distinguish the structure from a function call, unless the structure is special, or it occurs in a special context.

The equality test T_1 == T_2 is true if term T_1 and term T_2 are identical. The inequality test T_1 !== T_2 is the same as not T_1 == T_2. Note that two terms can be identical even if they are different terms stored in different memory locations. Also note that two terms of different types can be tested for equality, but they are never identical. The *unification* T_1 = T_2 is true if term T_1 and term T_2 are already identical, or if they can be made identical by instantiating the variables in the terms. The built-in T_1 != T_2 is the same as not T_1 = T_2. Note that among the four comparison operators ==, !==, =, and !=, only = can change the state of the variables in the compared terms.

2.2   Predicates and Functions

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable* rule

$$Head, \ Cond \Rightarrow Body.$$

and the *backtrackable* rule

$$Head, \ Cond \ ?\!\Rightarrow Body.$$

In a predicate definition, the *Head* takes the form $p(t_1, \ldots, t_n)$, where p is a predicate name, and n is the arity. The condition *Cond*, which is an optional goal, specifies a condition under which the rule is applicable. For a call C, if C matches *Head* (i.e., there exists a substitution θ such that $Head\theta = C$) and *Cond* succeeds, then the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites C into *Body*. If the used rule is non-backtrackable, then the rewriting is a commitment, and the program can never backtrack to C. However, if the used rule is backtrackable, then the program will backtrack to C once *Body* fails, meaning that *Body* will be rewritten back to C, and the next applicable rule will be tried on C. The backtrackable rule is semantically equivalent to:

$$Head \ ?\!\Rightarrow Cond, \ Body.$$

However, in-line tests that are written to the left of ?=> will be used by the compiler to index the rule.

The following defines the predicate member:

```
member(X, [Y|_]) ?=> X = Y.
member(X, [_|T]) => member(X, T).
```

Like the Prolog built-in member(X,L), this predicate can be utilized to check if X is a member of the list L, and it can also be utilized to retrieve an element of L through X by backtracking if X is a variable. Unlike Prolog's member(X,L), which can succeed an infinite number of times if L is a variable, the Picat definition can never succeed more times than the number of elements in L, since pattern-matching never changes call arguments.

A *function* is a special kind of a predicate that is defined by non-backtrackable rules. In a function definition, the $Head$ takes the form $f(t_1, \ldots, t_n) = Term$, where f is a function name and $Term$ is a result to be returned. If $Cond$ and $Body$ are both true, then they can be omitted together with the `=>` arrow.

The following gives two functions for reversing a list:

```
naive_reverse([]) = [].
naive_reverse([H|T]) = naive_reverse(T) ++ [H].

reverse(L) = reverse_aux(L, []).

reverse_aux([], Acc) = Acc.
reverse_aux([H|T], Acc) = reverse_aux(T, [H|Acc]).
```

For a list, if it is empty, then `naive_reverse` returns the empty list; otherwise, `naive_reverse` attaches the head `H` to the end of the reversed list of the tail `T`. The call `naive_reverse(L)` takes $O(n^2)$ time to reverse list `L` of length n. The function `reverse` calls `reverse_aux`, which scans the list while accumulating the reversed list in the second argument. The call `reverse(L)` takes linear time to reverse list `L`.

Picat, like functional programming languages, discourages the use of side effects in describing computations. All of the built-in functions in Picat's `basic` module are side-effect-free mathematical functions. Pure, side-effect-free functions are not dependent on the context in which they are applied. This purity can greatly enhance the readability and maintainability of programs.

Picat's dot notation makes calling a function look like calling a method on an object, as in `X.to_string().reverse()`. This *uniform function call syntax*[2] is convenient for chaining function calls in a readable way.

2.3 Loops and Comprehensions

Picat provides *loops* for describing repetitions and *comprehensions* for constructing lists and arrays. A `foreach` loop has the following general form:

```
foreach (E₁ in D₁, Cond₁, ..., Eₙ in Dₙ, Condₙ)
    Goal
end
```

The expression E_i `in` D_i is called an *iterator*, where E_i is an iterating pattern, and D_i is an expression that gives a compound value. Each $Cond_i$ is an optional condition on iterators E_1 through E_i. A loop statement forms a name *scope*: variables that occur in a loop, but do not occur before the loop in the outer scope, are local to each iteration of the loop.

A list comprehension has the following general form:

$$[Exp : E_1 \text{ in } D_1, Cond_1, \ldots, E_n \text{ in } D_n, Cond_n]$$

---

[2] http://dlang.org/spec/function.html.

where *Exp* is an expression, and the iterators and conditions have the same format as those used in the `foreach` loop. A list comprehension is a special functional notation for creating lists. It includes *Exp* as an element in the list for each possible combination of values in the iterators that satisfies the conditions. Like a loop, a list comprehension also forms a name scope.

An array comprehension has the following general form:

$$\{Exp : E_1 \text{ in } D_1, Cond_1, \ldots, E_n \text{ in } D_n, Cond_n\}$$

An array comprehension first creates a list, and then calls the function `to_array` to convert the list to an array.

The following gives an example that uses loops and comprehensions:

```
matrix_multi(A, B) = C =>
    C = new_array(A.length, B[1].length),
    foreach (I in 1..A.length, J in 1..B[1].length)
        C[I,J] = sum([A[I,K]*B[K,J] : K in 1..A[1].length])
    end.
```

The function `matrix_multi(A,B)` takes two matrices, `A` and `B`, that are represented as two-dimensional arrays, and returns the product `A × B`. All three of the variables `A`, `B`, and `C`, are non-local to the loop, because they occur before the loop. Note that, for an aggregate function, such as `sum` or `len`, that takes a list comprehension as the argument, the compiler generates a special function that computes the aggregate without actually building a list.

Loops are very convenient for scripting. The following gives an example program which recursively copies all of the files in a directory and its subdirectories to the current directory:

```
import os.

main =>
    WD = pwd(),
    flatten_dir(WD, WD).

flatten_dir(WD, Dir) =>
    Fs = listdir(Dir),
    foreach (F in Fs, F !==".", F !=="..")
        FullName = full_path(Dir, F),
        if directory(FullName) then
            flatten_dir(WD, FullName)
        else
            cp(FullName, full_path(WD, F))
        end
    end.

full_path(Dir, Name) =
    Dir ++ [separator()] ++ Name.
```

This program imports the `os` module, from which the built-ins `pwd`, `listdir`, `directory`, `cp`, and `separator` are used. The function `listdir(Dir)` returns

a list `Fs` of files and directories in the directory `Dir`. For each item `F` in `Fs`, if `F` is neither `"."` nor `".."`, then the program calls `full_path` to construct the full name `FullName` of `F`. If `FullName` is a directory, then the program recursively calls `flatten_dir` on the directory; otherwise, it copies the file to the `WD` directory.

2.4   Assignments and While Loops

Picat variables are *single-assignment*, meaning that once a variable is bound to a value, the variable cannot be bound again, unless the value is a variable or the value contains variables. In order to simulate imperative language variables, Picat provides the assignment operator `:=`. An assignment takes the form

$$LHS \ := \ RHS$$

where LHS is either a variable or an access of a compound value in the form $X[...]$. When LHS is a variable, the assignment does not actually assign the value of RHS to LHS. Instead, it creates a new variable for LHS to hold the value of RHS. After the assignment, whenever LHS is accessed in the body, the new variable is accessed. When LHS is an access in the form $X[I]$, the component of X indexed I is updated. This update is undone if execution backtracks over this assignment.

An assignment in the form `X[I] := RHS` has global side effects, since the compound term that is referenced by `X` is destructively updated, like an assignment in an imperative language. An assignment in the form `X := RHS`, where `X` is a variable, only has a side effect within the body of the rule in which the assignment occurs. Recall that the compiler introduces a new variable for `X` and replaces the remaining occurrences of X by the new variable. Variable assignments do not have cross-predicate or cross-function side effects.

An assignment makes it possible to use a variable to hold values at different stages during computation without inventing new variable names. With assignments, Picat is able to provide while loops that repeat under a condition. A `while` loop has the form:

```
while (Cond)
    Goal
end
```

As long as $Cond$ succeeds, the loop will repeatedly execute $Goal$. A `do-while` loop has the form:

```
do
    Goal
while (Cond)
```

A do-while loop is similar to a while loop, except that a do-while loop executes $Goal$ one time before testing $Cond$. In order for a while loop to make sense,

Goal must contain assignments that change some variables in *Cond*, unless the loop is meant to be an infinite loop.[3]

2.5   Constraint Modeling

Picat provides three solver modules, `cp`, `sat`, and `mip`, for modeling and solving constraint satisfaction and optimization problems (CSPs). As a constraint programming language, Picat resembles CLP(FD) [3]: the operators `::` and `notin` are used for domain constraints, the operators `#=`, `#!=`, `#>`, `#>=`, `#<`, `#<=`, and `#=<` are used for arithmetic constraints, and the operators `#/\` (and), `#\/` (or), `#^` (xor), `#~` (not), `#=>` (if), and `#<=>` (iff) are used for Boolean constraints. Picat supports several global constraints, such as `all_different/1`, `element/3`, and `cumulative/4`. In addition to intensional constraints, Picat also provides two predicates for expressing extensional constraints: `table_in/2` and `table_notin/2`.

The following gives a solution for the Fashion Police problem, which was used in GCJ Round 1 C 2016.[4] You have brought along J different jackets (numbered $1, \ldots, J$), P different pairs of pants (numbered $1, \ldots, P$), and S different shirts $(1, \ldots, S)$, $J \leq P \leq S$. Every day, you will pick one jacket, one pair of pants, and one shirt to wear as an outfit. You will be put into jail if you have worn the exact same outfit twice or if you have worn the same two-garment combination more than K times in total for some input K. Determine the maximum number of days that you will be able to avoid being taken to jail. The problem entails finding a maximum subset of outfits that satisfies the cardinality limit. For example, for $J = 1$, $P = 1$, $S = 3$, and $K = 2$, the answer is 2, because (1,1,1) and (1,1,2) are possible outfits, while adding the third outfit (1,1,3) to the list will violate the cardinality limit.

```
import util, sat.

main =>
    T = read_line().to_int(),
    foreach (TC in 1..T)
        [J,P,S,K] = [to_int(Token) : Token in read_line().split()],
        not not do_case(TC,J,P,S,K)
    end.

do_case(TC,J,P,S,K) =>
    L = {{Ij,Ip,Is} : Ij in 1..J, Ip in 1..P, Is in 1..S},
    N = J * P * S,
    Bs = new_array(N),
    Bs :: 0..1,
    sum(Bs) #=< J * P * K,    % pigeonhole principle
    foreach (R1 in 1..N)
        L[R1] = {Ij,Ip,Is},
        if S > K then
```

---

[3] It is possible to write an infinite loop as `while (true)` *Goal* `end`.
[4] https://code.google.com/codejam/contest/4314486/dashboard#s=p2\&a=2.

```
            Bs[R1] #=> sum([Bs[R2] : R2 in 1..N, L[R2] = {Ij,Ip,_}]) #=< K
        end,
        if P > K then
            Bs[R1] #=> sum([Bs[R2] : R2 in 1..N, L[R2] = {Ij,_,Is}]) #=< K
        end,
        if J > K then
            Bs[R1] #=> sum([Bs[R2] : R2 in 1..N, L[R2] = {_,Ip,Is}]) #=< K
        end
    end,
    solve([$max(sum(Bs))],Bs),
    printf("Case #%w: %w\n", TC,sum(Bs)),
    foreach (R in 1..N, Bs[R] == 1, L[R] = {Ij,Ip,Is})
        printf("%w %w %w\n", Ij,Ip,Is)
    end.
```

The `main` predicate first reads in an integer `T`, which is the number of test cases. For each test case `TC` in `1..T`, the body of the loop reads in `J`, `P`, `S`, and `K` in one line. The call `do_case(TC,J,P,S,K)` solves the case.[5]

The `do_case` predicate creates an array `L` of all possible outfits, computes the number of possible outfits `N`, and creates an array `Bs` of N Boolean variables. Each outfit is associated with one Boolean variable, which indicates whether the outfit is in the subset. The constraint `sum(Bs) #=< J * P * K` encodes the pigeonhole principle.[6]

The `foreach` loop ensures that no pair of garments occurs in outfits more than K times in the subset. For an outfit number `R1`, let `{Ij,Ip,Is}` be the outfit. The constraint

```
  Bs[R1] #=> sum([Bs[R2] : R2 in 1..N, L[R2] = {Ij,Ip,_}]) #=< K
```

ensures that, if the outfit `{Ij,Ip,Is}` is in the subset (`Bs[R1]` = 1), then the number of the jacket-pants pair `{Ij,Ip}` does not occur in the outfits more than K times in the subset. This constraint is only generated if `S > K`, because if `S ≤ K`, it is impossible to have more than K pairs of `{Ij,Ip}`. The `foreach` loop also generates cardinality constraints to ensure that the number of jacket-shirt pairs and the number of pants-shirt pairs do not exceed the limit.

The statement `solve([$max(sum(Bs))],Bs)` calls the SAT solver to solve the constraints such that the objective `sum(Bs)` is maximized.

GCJ problems normally require some amount of insight to solve, even for small datasets. This program is based on a straightforward model. Nevertheless, it solves the large dataset in 3 min, which is within the time limit of 8 min. This example demonstrates the use of Picat's language constructs, including arrays, loops, and list comprehensions, in modeling constraint problems.

3   Canonicalizing the Language Constructs

The Picat implementation adopts the virtual machine TOAM [17], which is a redesign of the Warren Abstract Machine [16] for fast software emulation. TOAM

---

[5] The double negation `not not` is used here to discard the generated constraints after the case is done.

[6] Since $J \leq P \leq S$, $\min(J \times P \times K, \ P \times S \times K, \ J \times S \times K)$ equals $J \times P \times K$.

provides instructions for encoding pattern-matching rules. An extended TOAM, which supports tabling and constraint propagation, is described in [17]. The Picat implementation reuses codes from the B-Prolog system, except the parser, the preprocessor, and many library built-ins. An early implementation of the SAT compiler, which translates high-level constraints into CNF, is given in [19].

Picat translates programs into a canonical form, which is further compiled into TOAM. This section describes the canonical form and the translation of Picat's different language constructs into the form. The compilation of the canonical form into TOAM is detailed in [17].

3.1   Canonical-Form Rules

A *canonical-form* rule takes one of the following forms:

> $Head$, $Cond$ => $Body$.
> $Head$, $Cond$?=> $Body$.

A canonical-form rule is different from the source-language rule in that *Cond* and *Body* do not include functions, comprehensions, loops, or assignments; all of these constructs are compiled away by Picat's preprocessor.

3.2   Transformation of Functions

Picat replaces a function call $f(t_1, \ldots, t_n)$ by a new variable V, and inserts a new predicate call $p(t_1, \ldots, t_n, V)$ before the goal in which the function call occurs. For a rule in the definition of f/n

$$f(t_1, \ldots, t_n) \; = \; Exp, \; Cond \texttt{ => } Body.$$

Picat translates it into the following predicate rule:

$$p(t_1, \ldots, t_n, V), \; Cond \texttt{ => } Body, \; V = Exp.$$

Picat employs an optimization to generate a tail-recursive rule if Exp is a list.

Consider, for example, the following function `conc`, which concatenates two lists:

```
conc([], Ys) = Ys.
conc([X|Xs], Ys) = [X | conc(Xs, Ys)].
```

Picat translates it into the following predicate:

```
conc_p([], Ys, Zs) => Zs = Ys.
conc_p([X|Xs], Ys, Zs) =>
    Zs = [X|Zs1],
    conc_p(Xs, Ys, Zs1).
```

The Picat compiler incorporates the *tail-recursion optimization*, which translates tail-recursive deterministic predicates into iteration. A call to `conc_p` only allocates one frame on the stack, no matter how long the list is.

3.3 Transformation of Comprehensions

Picat translates a list comprehension into a `foreach` loop that uses `:=` to accumulate values. The list comprehension

$$[Exp \ : \ E_1 \text{ in } D_1, \ Cond_1, \ \ldots, \ E_n \text{ in } D_n, \ Cond_n]$$

is replaced by a new variable `L`, and the following statements are inserted into the context:

```
L = Tail,
foreach (E_1 in D_1, Cond_1, ..., E_n in D_n, Cond_n)
    Tail = [Exp|NewVar],
    Tail := NewVar,
end,
Tail = []
```

Initially, `Tail` is a free variable. In the body of the loop, the call

```
Tail = [Exp|NewVar]
```

binds `Tail` to the term `[Exp|NewVar]`, and the assignment `Tail := NewVar` lets `Tail` reference the new tail. After the loop, the call `Tail = []` binds `Tail` to `[]`, completing the list. An alternative translation is possible, which begins with an empty list, and attaches each value to the end of the list. However, this translation is not efficient, since it takes linear time to add a value to the end of a list.

List comprehensions that occur in aggregate functions, including `sum(L)`, `min(L)`, `max(L)`, and `len(L)`, are compiled in such a way that an aggregate value is computed, rather than a list. For example, the function call

```
sum([f(I) : I in 1..100])
```

is replaced by a new variable `Sum`, and the following statements are inserted into the context:

```
S = 0,
foreach (I in 1..100)
    S := S + f(I)
end,
Sum = S
```

3.4 Transformation of Pure `foreach` Loops

Picat translates loops into tail-recursive predicates. Consider pure `foreach` loops that do not contain assignments. Without loss of generality, consider a `foreach` loop that has only one iterator:

```
foreach (E in D)
    Goal
end
```

The general form of the `foreach` loop can be converted into this form by introducing if-then statements and nested loops into the loop goal. For a loop statement that has nested loops, the inner-most loop is transformed first.

Let V_1, V_2, ..., V_n be the global variables in *Goal*, i.e., variables that occur before the loop in the context. If D is a list, then the loop is replaced by a predicate call in the form $p(V_1, V_1, ..., V_n, D)$, where p is a newly generated predicate:

```
p(V₁,V₁,...,Vₙ,[]) => true.
p(V₁,V₁,...,Vₙ,[E|T]) => Goal, p(V₁,V₁,...,Vₙ,T).
```

Note that local variables in *Goal* are not passed to predicate p, and are naturally localized. If D is an array or a range in the form `LB..Step..UB`, then the generated predicate takes extra arguments for iterating over the array or the range using recursion.

3.5   Transformation of Assignments

An assignment that updates a compound value is transformed into a built-in predicate called `segarg`. This subsection shows how to transform assigned variables.

For an assignment $LHS := RHS$ that occurs in a conjunction of goals, Picat introduces a new variable for LHS that holds the value of RHS. For example, consider the following rule:

```
test => X = 0, X := X + 1,  X := X + 2, write(X).
```

Picat creates a new variable, say `X1`, to hold the value of `X` after the assignment `X := X + 1`. Picat replaces `X` by `X1` on the LHS of the assignment. All occurrences of `X` after the assignment are replaced by `X1`. When encountering `X1 := X1 + 2`, Picat creates another new variable, say `X2`, to hold the value of `X1` after the assignment, and replaces the remaining occurrences of `X1` by `X2`. When `write(X2)` is executed, the value held in `X2` is printed. After preprocessing, the rule is translated into the following:

```
test => X = 0, X1 = X + 1, X2 = X1 + 2, write(X2).
```

For an assignment that occurs in if-then-else, Picat introduces a new predicate. Consider the following example:

```
go(Z) =>
    X = 1, Y = 2,
    if Z > 0 then
        X := X * Z
    else
        Y := Y + Z
    end,
    println([X,Y]).
```

Picat translates the program into the following:

```
go(Z) =>
    X = 1, Y = 2,
    p(X, Xout, Y, Yout, Z),
    println([Xout,Yout]).

p(Xin, Xout, Yin, Yout, Z), Z > 0 =>
    Xout = Xin * Z,
    Yout = Yin.
p(Xin, Xout, Yin, Yout, Z) =>
    Xout = Xin,
    Yout = Yin + Z.
```

One rule is generated for each branch of the if-then-else statement. For each variable V that occurs on the LHS of an assignment that is inside of the if-then-else statement, predicate p is passed two arguments, Vin and Vout. In the above example, X and Y each occur on the LHS of an assignment. Therefore, predicate p is passed the parameters Xin, Xout, Yin, and Yout.

Similarly, for an assignment that occurs in a loop, Picat passes two variables to the predicate for the loop: one variable holds the value before the loop goal is executed, and the other holds the value after the loop goal is executed. Consider the following example:

```
sum_list(L, Sum) =>
    S = 0,
    foreach (E in L)
        S := S + E
    end,
    Sum = S.
```

Picat translates the program into the following:

```
sum_list(L, Sum) =>
    S = 0,
    p(L, S, Sout),
    Sum = Sout.

p([], Sin, Sout) =>
    Sout = Sin.
p([E|T], Sin, Sout) =>
    St = Sin + E,
    p(T, St, Sout).
```

In addition to the list L, Picat passes arguments S and Sout to predicate p. Note that only the global variables that occur within the loop are passed to p.

4 Related Work

The canonical-form language that is used in the Picat compiler is called *matching clauses* in B-Prolog [17]. This canonical form narrows the gap between the high-level constructs and the underlying virtual machine.

Functions are naturally a basic notion in functional logic languages, such as Curry [5]. Picat, like other logic programming languages, such as Mercury [9] and Ciao [7], provides functions as a syntax extension. Picat has limited support for higher-order predicates and functions, and does not support lambda expressions. The use of higher-order calls in Picat is discouraged because of the overhead.

Classic functional and logic languages rely on recursion and higher-order facilities to describe repetitions. Many modern languages, such as F#[7] and OCaml[8], provide looping constructs. Picat's `foreach` loop is similar to B-Prolog's `foreach` loop [17], which was inspired by logical loops in ECLiPSe [12]. In ECLiPSe, variables are assumed to be local to each iteration, unless they are declared global. In B-Prolog, variables are assumed to be global to all of the iterations, unless they are declared local. In contrast, Picat adopts a simple and clean scoping rule for variables, which renders the declaration of local or global variables unnecessary.

The list comprehension, which can be traced back to SETL [13], was made popular by Haskell. The optimization that computes a value instead of creating a list when a comprehension is immediately fed into an aggregate function, such as `sum`, is a special application of the idea of deforestation [14]. The same idea is employed in compiling loops whose iterators contain the range `..` and the `zip` function. The Picat compiler does not apply deforestation to user-defined functions or built-in functions in other contexts.

It is rare for declarative languages to provide assignments. An assignment of the form `S[I] := RHS` is similar to the `setarg` built-in in Prolog. An assignment of the form `X := RHS`, where `X` is a variable, is translated into unification at compile time. The transformation rules that eliminate assignments are employed in building the static single assignment form (SSA) for imperative programs, which simplifies program analysis and compilation [2]. The Picat compiler introduces a new predicate for every branching statement that contains assignments, even for if-then-else, which could be compiled inline. This makes it unnecessary to introduce a phi function [2] when branches merge.

There are abundant examples that demonstrate the usefulness and convenience of Picat's language constructs for modeling. In [4], several examples are given for GCJ problems.

5   Conclusion

This paper has presented the Picat language, and has shown how to compile Picat's high-level language constructs into canonical-form pattern-matching rules. The high-level language constructs give Picat flexibility and brevity needed for scripting. Picat provides a comprehensive box of tools for describing and solving combinatorial search problems. The high-level constructs also facilitate modeling with these tools.

---

[7] http://fsharp.org/.

[8] http://ocaml.org/.

References

1. Colmerauer, A.: Equations and inequations on finite and infinite trees. In: Proceedings of FGCS, pp. 85–99. ICOT (1984)
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form, the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
3. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The constraint logic programming language CHIP. In FGCS, pp. 693–702 (1988)
4. Dymchenko, S., Mykhailova, M.: Declaratively solving Google code jam problems with Picat. In: Pontelli, E., Son, T.C. (eds.) PADL 2015. LNCS, vol. 9131, pp. 50–57. Springer, Cham (2015). doi:10.1007/978-3-319-19686-2_4
5. Hanus, M.: Functional logic programming: from theory to curry. In: Programming Logics, pp. 123–168 (2013)
6. Van Hentenryck, P.: Constraint and integer programming in OPL. INFORMS J. Comput. **14**, 2002 (2002)
7. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of Ciao and its design philosophy. Theor. Pract. Logic Program. **12**(1–2), 219–252 (2012)
8. Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artif. Intell. **2**(3–4), 227–260 (1971)
9. Mercury. http://www.mercurylang.org/
10. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Principles and Practice of Constraint Programming, pp. 529–543 (2007)
11. Schimpf, J.: Logical loops. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 224–238. Springer, Heidelberg (2002). doi:10.1007/3-540-45619-8_16
12. Schimpf, J., Shen, K.: Eclipse-from LP to CLP. Theor. Pract. Logic Program. **12**(1–2), 127–156 (2012)
13. Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: Programming with Sets - An Introduction to SETL. Springer, New York (1986)
14. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1990)
15. Warren, D.H.D.: High-order extensions to Prolog - are they needed? Mach. Intell. **10**, 441–454 (1982)
16. Warren, D.H.D.: An abstract Prolog instruction set. Technical note 309, SRI International (1983)
17. Zhou, N.-F.: The language features and architecture of B-Prolog. Theor. Pract. Logic Program. **12**(1–2), 189–218 (2012)
18. Zhou, N.-F., Barták, R., Dovier, A.: Planning as tabled logic programming. In: Theory and Practice of Logic Programming (2015)
19. Zhou, N.-F., Kjellerstrand, H.: The Picat-SAT compiler. In: Gavanelli, M., Reppy, J. (eds.) PADL 2016. LNCS, vol. 9585, pp. 48–62. Springer, Cham (2016). doi:10.1007/978-3-319-28228-2_4
20. Zhou, N.-F., Kjellerstrand, H., Fruhman, J.: Constraint Solving and Planning with Picat. Springer, Heidelberg (2015)