IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

`LogicAPI`: Logic for Python

Author:
Zimu Qin

Supervisors:
Prof Susan Eisenbach
Dr Fariba Sadri

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2017

Abstract

The aim of the project is to develop a Python library to enable the logic programming paradigm. Although there exists such libraries already, they are impractical to solve the complex logic programming problem due to the lack of features and the poor performance. The new library called `LogicAPI` is implemented with the aid of object oriented paradigm. It is easy to use and extend the system which can interact with Python functions by using the given interfaces. By using the system, a solver for the river-crossing puzzle is implemented and used to visualize the solutions with the help of other graphics/animation library.

Acknowledgments

I would like to thank my supervisors, Prof Susan Eisenbach and Dr Fariba Sadri sincerely for helping me and giving me advices whenever I need. I also want to express appreciation to my friend Duyang Zhou for his support when I met with difficulties in the project. Without their help, the project would never be possibly done.

Contents

Chapter 1

Introduction

In this chapter, we would give a general introduction of the project, including the motivation, the goal, and the result of the project.

1.1 Motivation

A logic programming language, such as Prolog is ideal to solve many logic-related tasks such as searching and planning, because it is declarative, which means we only need to sufficiently describe the problem, not implement algorithms in explicit steps [19]. It is very useful for theorem proving, automated planning and searching, etc.[11] However, the functionalities are restricted when comparing to other popular programming languages such as Input/Output and file handing, and due to the significant difference between logic programming and others in syntax, logic programming is also not very easy to learn for beginners. Thus, we want to extend the power of logic programming to another popular language which could provide the functionalities that is lacking in logic programming.

Python is a very popular programming language, but it does not support the logic programming paradigm currently. It provides many useful features like built-in functions and libraries, dynamic typing and high-level data types. It supports multiple programming paradigms including object-oriented, imperative, functional programming, and procedural styles [26]. With many features introduced by the built-in and external libraries, Python is a powerful language that is easy to learn and write. However, without the support for the logic programming paradigm, doing logic in Python is pretty tricky because Python is imperative, which means coders have to implement algorithms in explicit steps in Python. Therefore, combining these two could be very useful, and although there exists several systems which would be introduced in chapter 3, they are not very useful in many cases and impractical to accomplish complex tasks. Therefore, we want a new system that enables us to introduce the logic programming paradigm to Python, and use it for complex tasks later.

1.2 Objective

The aim of the project is to design and implement an external library of Python which enables the logic programming paradigm. Such system should be able for the users to do complex tasks combining logic reasoning and the power of Python. For example, it should be able to visualize the solutions of the river-crossing puzzle which is introduced in section 2.4, in which the solver is implemented by using the logic programming paradigm, and the visualization part is implemented with the aid of Python external libraries to do graphics and animations. There are existing systems that allow programmers to do logic programming in python by using an extensive library, but at this moment, there is not one ideal for the coders. The new system to be developed should be easier to use, and by using it, it should be trivial to implement a visualization of solving the river-crossing puzzle, which would be discussed in detail later.

1.3 Achievement

The finished library `LogicAPI` is designed following the object-oriented paradigm. It allows user to define the knowledge base of the system, and pose queries on a functional term or a conjunction of functional terms. It provides an easy way for users to use Python functions in the logical clauses and an interface for users to get the solutions computed by the resolution system. By using the system, we implemented a visualization of solving the river-crossing puzzle. For the result, please see chapter 6. Compared to pyDatalog, which is another similar system introduced in section 3.4, the performance of the resolution engine is much higher. It also has features like cut operators that are not available in any other similar systems. Although there are still many features that have not been added, the system is implemented in a way that it could be easily extended, so the system could be further improved.

Chapter 2

Background

In this chapter, some background knowledge would be introduced, including some basic concepts in logic programming, some features of Python which would be used later, and the description of the river-crossing puzzle.

2.1 Logic Programming

Logic programming is a kind of programming paradigm, by using which the users can declare facts and rules in the code, and then pose queries to get the solution from the description of the problem. [20]

2.1.1 Basic Terms

Definitions

Term A term in logic programming can be either

- a constant,
- a variable,
- or a functional term

In Prolog, a constant is either an atom or a number. An atom in Prolog can be either

- a sequence of characters started with a lower case English letter with no space in it, e.g. char_Seq
- a string, e.g. 'this is a string'
- a sequence of special characters$(+, *, -, \backslash, / \ldots)$
- or one of the special atoms: $[], \{\}, ;, !$

In Prolog, a variable is a sequence of characters started with a upper case English letter or the underscore character, e.g. $Variable$ and $\_internal\_Variable$.

In Prolog, a functional term is defined with a functor and a list of arguments. For example, to represent the term which apply the functor f to the arguments X and Y, $f(X, Y)$ is used. Note that the arguments of the functional term could be another functional term, thus it is a recursive type. Lists are a functional term whose functor is a dot. A list's arguments are its head and tail. Thus a list of $[1, 2, 3]$ could also be represented as $.(1, .(2, .(3, [])))$.

2.1.2 Unification

Unification is an important part in logic programming language. It is the process of determining if two terms can be made equaled by substituting their variables.

First unification algorithm was given by Robinson in 1965, but it was inefficient due to the occur check to ensure the soundness of the resolution strategy. [23] Thus, the algorithm was further improved later to make the language usable. A method that many Prolog system used is to simply drop the occur check to make the unification process more efficient.

Substitution A substitution σ is a finite set of pairs $(variable, term)$ such that all the variables on the left side are distinct from each other. [10]

For example,

$$\sigma = \{(X, Y), (Z, 1)\} \tag{2.1}$$

We say X is bound to Y, and Z is bound to 1. Since Y does not appear on the left side of the substitution, it is a free variable.

To apply substitution to a term, every variables in the term needs to be substituted if applicable. Use a functional notation, e.g. $\sigma(f(X))$ to represent applying σ to $f(X)$.

A unifier of two terms is a substitution that two terms equaled after applied with it. For example,

$$term1 = f(X, Y), term2 = f(Z, 1) \tag{2.2}$$

Then $\sigma = \{(X, 1), (Y, 1), (Z, 1)\}$ is a unifier of $term1$ and $term2$, as they both equal to $f(1, 1)$ after applied with σ.

A unifier σ of two terms are called the most general unifier if and only if for all unifiers θ, there exists a substitution λ such that $\theta = \lambda \circ \sigma$, i.e. the most general unifier is a subset of every unifier. For example, for the above terms, the most general unifier should be $\sigma = \{(X, Y), (Z, 1)\}$. The unifier could also be seen as instantiations of variables.

Robinson's Unification Algorithm

The unification algorithm given by Robinson in 1965 can decide whether two terms are unifiable, and returns the most general unifier. The algorithm is described as below [10]:

```
Given: two terms $t_1$ and $t_2$.
Result: a pair $(bool, \sigma)$ such that:
        -$bool = true$ if and only if the two terms are unifiable.
        -if $bool = true$ then $\sigma$ is the most general unifier for $t_1$ and $t_2$.
Algorithm:
        if one of the two terms is a variable $x$, call the other term $t$.
          then if $x = t$
                  then $bool := true$; $s := \emptyset$
                  else if $occur(x, t)$
                          then $bool := false$
                          else $bool := true$; $\sigma := \{(x, t)\}$
                          endif
                  endif
          else    let $t_1 = f(x_1, ..., x_m)$ and $t_2 = g(y_1, ..., y_n)$
                  if $f \neq g$ or $n \neq m$
                          then $bool := false$
                          else $i := 0$
                               $bool := true$;
                               $\sigma := \emptyset$;
                               while $i < n$ and $bool$ do
                                       $\imath := i + 1$;
                                       $(bool, \sigma_1) := unify(\sigma(x_k), \sigma(y_k))$;
                                       if $bool$ then $\sigma := \sigma_1 \circ \sigma$ endif
                               endwhile
                  endif
        endif
```

Figure 2.1: Robinson's Unification Algorithm

Note that because of the function $occur$, each time we need to recursively check if the variable exists in the term. If the term is a complex term, and the depth of it is great, then it would cause the algorithm to run very slowly. Although it is added to make sure the resolution engine is sound, it would make the language much less inefficient than others, and therefore not very practical. Thus, for most Prolog systems, the occur check is simply dropped to ensure a more efficient language. Further work can be done to ensure the soundness of the engine, but it would not be discussed here.

2.1.3 The Knowledge Base

The knowledge base of a logic programming is essentially a database or collection of clauses. A program is a user-defined knowledge base , in which we could make queries. A clause can be either a fact or a rule. A fact is represented as

```
H.
```

A rule is represented as

```
H :- B_1 , B_2 , . . . , B_N .
```

It means ”H if B_1 and B_2 and ... and B_N”.

We could see the knowledge base as a collection of facts and rules, based on which the system is able to resolve a goal.

2.1.4 Resolution Strategy

The execution of a logic program depends on the resolution proof method [4]. The particular resolution strategy used by Prolog is called SLD-resolution (Selective Linear Definite clause resolution). [18] The systems of Prolog always select the clauses whose head can be unified with the queried term in the order of their appearance. Then, the system would make queries for each of the terms in the body. If any of them fails, then go to the last choice point and try to make another choice. This process is called ”backtracking”. [14] Therefore, the process of query can be seen as depth-first search, and the execution process can be represented as a search tree.

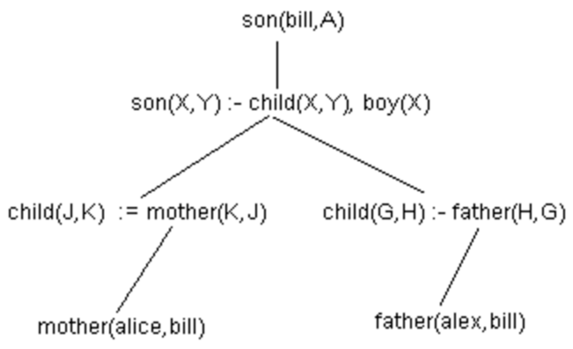

son(bill,A)
son(X,Y) :- child(X,Y), boy(X)
child(J,K) := mother(K,J)
child(G,H) :- father(H,G)
mother(alice,bill)
father(alex,bill)

Figure 2.2: An example of search tree in logic programming [21]

2.2 Python

Python is a very widely-used interpreted general-purpose programming language [3]. Its syntax is designed in a way to ensure readability (e.g. indentations rather than blocks) and allow coders to complete the same task in fewer lines of codes comparing to other programming languages like C and Java by sacrificing some performance. [24] It is dynamically-typed and supports many programming paradigms like functional programming and object-oriented programming. Unfortunately, until now, the logic programming paradigm is not yet supported. Python is designed to be a very easy-to-extend language [1], so we could extend Python with the logic programming paradigm as an external library easily.

2.2.1 Lists and Tuples

Lists are a fundamental data type in Python. They are used to store a collection of values which are assigned to a non-negative integer called index in order to provide access the values. They are represented as comma-separated values in a pair of square bracket. [8]
The underlying structure of the list uses an array of objects of variable lengths, instead of linked list. The resizing mechanism of it makes sure that the underlying array would not need to grow its size very often when the length of the list should grow. Except for getting or modifying an item, and getting the length of a list, appending a new item to the end, or popping the last item of the list takes only amortized $O(1)$ time in the worst case. Extending the second array to the end of the first array takes amortized $O(k)$ time in the worst case, where k is the length of the second array. [17]
Tuples are represented as as comma-separated values in a pair of round bracket. The underlying structure is very much the same as the lists, but tuples are immutable, which means the tuples cannot be changed after they are instantiated. Thus, we are unable to modify the values in it. It is possible to append an item or extend a tuple to a tuple by copying itself first before doing the operation, in order to make sure the original tuple stay the same.

2.2.2 Dictionaries

Dictionaries in Python are implemented by using hash tables which are an array whose indexes are the output of an underlying hash function whose input is the key of the dictionary. The values stored in the array are the values of the dictionary. [9] It could map keys to values if the keys in the dictionary are immutable, such as integers, strings or tuples. To define a dictionary, we could write:

```
x = {}
```

To add a key-value pair (1, 'one') to the dictionary, we use the following syntax:

```
x[1] = 'one'
```

Later we could use $x[1]$ to check the corresponding value in the dictionary. We could also delete an element in the dictionary using the keyword del:

```
del x[1]
```

The representation of a dictionary is comma-separated pair in the form $Key : Value$ surrounded by curly braces.[7] For example:

```
{1 : 'one', '2' : 'two'}
```

2.2.3 Generators

Generator functions are a special feature in Python that allows users to declare functions which return a generator object consisting of the code of the function and the current execution state of the function. In this way, the return value can behave like an iterator [9]. The keyword *yield* is used to suspend the function execution and released if the remaining return values are to be generated.

For illustrate this, we define a generator function *nat* that generates all the natural numbers:

```
def nat():
        i = 0
    while True:
        yield i
        i += 1
```

2.2.4 Operators

In Python, it is not possible to define new operators, as Python does not provide you the functionality. However, for the objects, some of the default operators can be overloaded. Below are some examples: [5]

Operator	Function	Syntax
Positive	`__pos__(a)`	`+ a`
Negative	`__neg__(a)`	`- a`
Addition	`__add__(a, b)`	`a + b`
Subtraction	`__sub__(a, b)`	`a - b`
Multiplication	`__mul__(a, b)`	`a * b`
Division	`__div__(a, b)`	`a / b`
Bitwise And	`__and__(a, b)`	`a & b`
Bitwise Inversion	`__invert__(a)`	`~a`
Indexing	`__getitem__(obj, k)`	`obj[k]`
Equality	`__eq__(a, b)`	`a == b`
Inequality	`__ne__(a, b)`	`a != b`
Less Than or Equal to	`__le__(a, b)`	`a <= b`
Less Than	`__lt__(a, b)`	`a < b`
Greater Than	`__gt__(a, b)`	`a > b`
Greater Than or Equal to	`__ge__(a, b)`	`a >= b`

In order to get the operators as objects of functions, we could use the library *operator*.

2.2.5 Metaclass

Metaclass is a special kind of class in Python. Unlike a super class in which you cannot control the behavior of the methods in the subclasses, and subclasses could override the methods defined in their super classes, a metaclass is used to modify the behavior of other classes. In Python, a metaclass could be created as a subclass of the class *type*. For example,

```
class MyMetaClass(type):
```

We could override the function *__init__* to modify the behavior of other classes' initializer. For example, we could add another field x to the object while initialization:

```
    def __init__(self, *args, **kwargs):
        type.__init__(self, *args, **kwargs)
        self.x = 1
```

For classes that need to use $MyMetaClass$ as a metaclass, just pass the class object to the field called *__metaclass__*. For instance, we could create a new class A:

```
class A(object):
        __metaclass__ = MyMetaClass
```

During the initialization of an instance of A, the attribute x would be automatically added to the instance.

2.2.6 Mixin Class

Mixins are a concept that coders write methods in a mixin class that other classes could reuse these methods by gaining the access of the mixin class. In Python particularly, it can be done by inheritance. For example, in Python, we could define a class with all the methods needed called $Mixin$, user can obtain all the methods in another class without knowing any detail of the underlying implementation, as just write:

```
class A(Mixin): pass
```

2.3 Python vs Logic Programming

Although the styles of programming in Python and logic programming are quite different, there are some similarities in them. Both Prolog and Python are dynamically typed, which means that until runtime, the variables in programs will not be checked for their types. They both support for high-level data types.

Python is imperative while logic programming is declarative as discussed before. The searching in logic programming is automated and does not need to be stated in the code explicitly, while in Python, the searching method must be well-stated explicitly in the code, thus is a bit more indirect to be implemented.

2.4 River-Crossing Puzzle

A river-crossing puzzle describes the problem to transports objects from one river bank to another, where there are restrictions on which objects and the number of objects to transport at the same time. [22] The Wolf, the Goat, and the Cabbage Problem is one famous variation of the puzzle. It describes the problem as follows: A farmer is traveling with a wolf, a goat and a cabbage. They need to cross a river by a boat which can only transport two objects at a time. The wolf would eat the goat if the farmer is not present. Similarly, the goat would eat the cabbage if the farmer is not present. The farmer needs to come up with a plan to successfully transport all of them to the other river bank safely without anyone being eaten. [12]

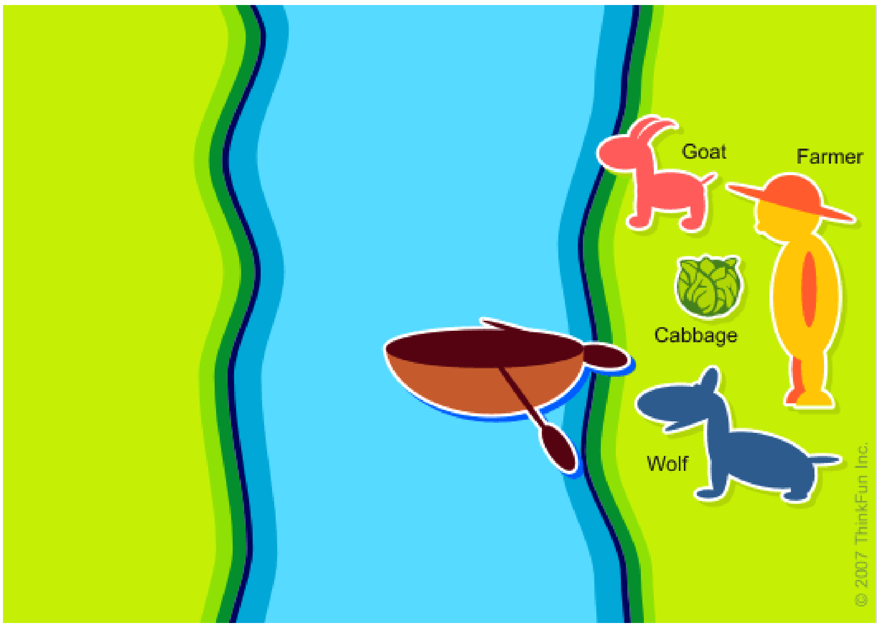


Figure 2.3: The river-crossing puzzle [6]

Chapter 3

Related Work

In this section, we will have a look at the existing systems enabling logic programming in Python, and evaluate the strengths and weaknesses of them.

3.1 Prolog in Python

It is a project of an external library of Python allowing coders to do logic programming that is fully implemented in Python. [21]
Some important classes are:

- `Term`: A Prolog data structure in the form of `x(y, z, ...)`.
- `Rule`: A Prolog predicate in the form of `term -: term, term, ...`.
- `Goal`: A stack of goals in order to implement the backtracking mechanism of Prolog.

Some important functions are:

- `unify(srcTerm, srcEnv, destTerm, destEnv)`: Unifies the source term and environment with the destination ones.
- `search(Term)`: Searches the given term.
- `eval(term, env)`: Given the current environment, evaluate the given term.

Advantages

The advantage of this system is that the implementation is quite straightforward and very easy to understand. It only contains about 200 lines of Python codes.

Disadvantages

There are not many features in the system. It just gives a simplified version of the resolution engine, but it is not practical to use it with Python. Another disadvantage of the system is the poor performance. Because it is implemented in Python and in order to keep the program simple, the functions are implemented in a naive way, and there is no optimization done. As a result, it could not be used in real world programming. Another disadvantage is that it does not support the automation like Prolog does, and it is not that declarative. Thus, it is very impractical to use it for logic programming in Python, though it is a good start.

3.2 PyLog

PyLog is a first-order logic library in Python, allowing coders to do logic programming. [15]
Some important classes are:

- `Term`: To define a term in Python, a functor needs to be defined using mixin class as `class f(Term):  pass`.
- `Var`: Because unlike Prolog, it is impossible to use a variable if it is not initialized, an instance of variable needs to be generated first: `X = Var()`.
- `Atom`: Atoms are the instances of Python objects.

Advantages

The size of the library is relatively small with about 600 lines of Python codes (including comments). By using the grammars described above, it tends to blur the boundary between Python programming and logic programming.

Disadvantages

One disadvantage of PyLog is that it does not have enough documentation to introduce all the grammars and features that were implemented, which really makes it hard to learn and use. There is only a very brief page giving some simple examples which does not help much. The functionality is quite limited since we have to build the goal stack and unify the term by ourselves, which means it is not that declarative and does not have the automation as Prolog does. The implementation seems incomplete as well, as the described feature of PROLOG engine (translates Prolog to Python) is not found.

3.3 PySWIP

PySWIP is a Python - SWI-Prolog bridge featuring an incomplete SWI-Prolog foreign language interface, a utility class that enables to query SWI-Prolog in Python programs. It also includes a functionality to register the python functions as Prolog predicates through the foreign language interface. [25]

Advantages

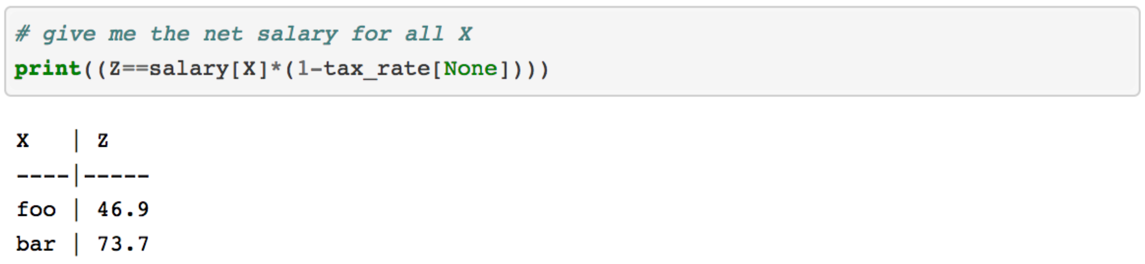
The advantage of the PySWIP is obvious. Because it is a bridge between SWI-Prolog and Python, you could do almost everything with it as in SWI-Prolog and get the result of the query back to Python. The feature to register a Python function as a Prolog predicate is very helpful, as it made calling the python function within the Prolog code possible.

Disadvantages

First of all, it depends on SWI-Prolog, so the users must install SWI-Prolog first. If SWI-Prolog is updated, for example, the compatibility could be a issue. Thus, we definitely would not like a library or interpreter to depend heavily on others. The performance could be restricted too. Another disadvantage is that the variable and the functions/predicates are not shared in the two languages. Although we have the feature of registering Python functions, we could not do the other way around. Thus, it is hard to get the result calculated by Prolog in real-time. If we want all the solutions imported to Python at once, in the other words, if we do not need the data in real-time, PySWIP would do the task perfectly.

3.4 PyDatalog

Adds the logic programming paradigm to Python's extensive toolbox, in a pythonic way. [2] In order to use terms in Datalog, they have to be created by calling the function `create_term(String)`, and feed in the terms in the form of string. The result of the query would be given in a table, listing all the possible solution.



```
# give me the net salary for all X
print((Z==salary[X]*(1-tax_rate[None])))
```

```
X   | Z
----|-----
foo | 46.9
bar | 73.7
```

Figure 3.1: An example of PyDatalog execution [2]

Advantages

It has many more features than other systems, and a well-written tutorial to lead through. Some optimization is done so the performance is better. For example, intermediate results are memorized to speed up the execution by avoiding duplicate work [2]. Adding links between nodes could be helpful since tree/graph problem is very common in logic programming. The documentation is well-written and easy-to-read with many examples given.

Disadvantages

One of the disadvantages is that Datalog is more limited than Prolog. For example, in Datalog [13]:

- it is not allowed to have a complex term as an argument of predicates such as `x(y(z))`.
- the 'cut' operator does not exist, since Datalog is truly declarative, so it should produce the same result no matter the order of the clauses.
- certain stratification restrictions are imposed on the use of negation and recursion.

In conclusion, it is a logic programming language that fits more for data query, like SQL does, but when it comes to more complicated planning and searching problem, it might not be powerful enough.

Chapter 4

Requirements & Design

In this chapter, we would talk about the requirements of features for the system we are building, and the design principles of the syntax and the overall structure of the system.

4.1 Requirements

The requirements of the system include:

1. It should be easy for users to define logical terms and clauses.
2. Users should be able to define them in a similar if not exactly the same style.
3. Users should be able to easily integrate Python codes in the system, so that they could mix Python programming with logic programming.

The system in the end should be able to complete tasks like visualizing the solving process of river-crossing puzzle, which combines the strength of Python and logic.

4.2 Design

4.2.1 Syntax

For the syntax of the system, it should be the best if we could follow the usual styles of logic programming. The variables and constants can be used in the same way as in Python. To design the structure for the functor of a predicate, there are two ways to get the same syntax in logical functional term. One way is to define it as a function, so that a functional term can be seen as the output of executing the function. Another way is to define it as a class, so that a functional term can be seen as the instantiation of an object. We choose the later one because by using the idea of inheritance and mixins, users could make a subclass of the class we define, and there is no need for the users to define any functionality of the class because everything would be inherited directly. To design the syntax of logical clauses is a bit different.

Facts

In Prolog, facts can be written like this:

```
H.
```

However, in Python, as we mentioned above, the functional term can be seen as the instantiation of an object. Thus, if we just put a functional term in a line, it could not added to the knowledge base automatically. We need an operator to perform the process of recording the clause to the knowledge base. In Python, we have the positive operator $+$ that can be overridden, which is a reasonable choice enabling users to define facts like this:

```
+ H
```

Rules

There is no way for us to create in the same style as in general logic programming languages. One reason for that is Python does not allow users to define their own operators, which means we could not have the ”if” operator $:-$ defined in Python. The behavior of the ”and” operator $,$ cannot be modified as well. Therefore, the only thing we could do is to use the operators already defined in Python and can be overridden. We use the ”less than or equal to” operator $<=$ to express the behavior of ”if” operator, and we use the ”and” operator $\&$ to replace $,$ between functional terms in the body of the clauses in logic programming. For example, in Prolog, we write rules in the following style:

```
H :- B_1, B_2, B_3.
```

In the system we design, the style of the rules will be like this:

```
H <= B_1 & B_2 & B_3
```

List

Lists, as a special functional term in Prolog, is represented in the style similar to Python. If the list is already determinate, which means the length of the list is fixed so it does not have a part unknown for the rest of the list, then it is represented the same as Python lists. For example, a list could be

```
[1, 2, 3, 4, 5]
```

However, in logical programming, the later part of the list might be a variable, which means that it is unknown at the moment. In the syntax of Prolog, a special symbol — is used to split the first known items and the unknown later part. This is very important for pattern matching on lists. For example, there might be a list that we know the first and the second element is 1, and the remaining list is unknown and denoted to the variable X.

```
[1, 1 | X]
```

It is impossible to define such syntax in Python, because as we mentioned above, we are not allowed to define our own operators. Therefore, we use another syntax which is more ”Pythonic”:

```
[1,1]+X
```

In this way, we only need to override the ”add” operator.

4.2.2 Structure

The structure of the system was designed in object-oriented way by using inheritance. The following diagram shows the relations of some basic classes:

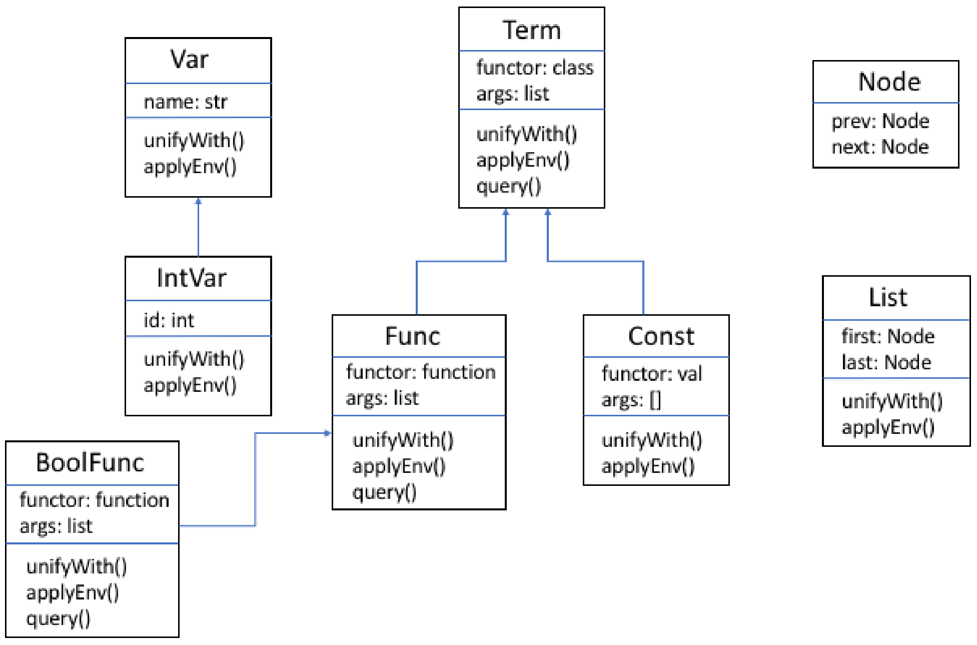


Figure 4.1: The UML class diagram of the object-oriented design

The structure of the system would be further introduced in the next chapter.

Chapter 5

Implementation

In this chapter, we would talk about the implementation of `LogicAPI`. This chapter mainly includes defining logic terms, implementing Robinson's unification algorithm, knowledge base creation, providing querying strategy and communication interface between Python and `LogicAPI`.

5.1 Basic Terms

First, we would define the basic logical terms in Python, as they are the most fundamental concepts in logic programming. The basic logic terms include variables, functional terms and constants.

5.1.1 Variable

Logical variables are very different as they would be instantiated during the process of substitution given the general unifier computed by unification algorithm, while Python variables must be assigned to a value before it is used. Thus, unlike in logical programming, we have to initialize a logical variable with its string representation stored, and then it could be used later for instantiation when applying the substitution. Thus, we create a new class of object called Var to represent the logical variables in the system.

```
class Var(object):
```

__init__ is called when a Var is created given the string representation.

```
    def __init__(self, name):
```

If the given representation is not a string, then it should be considered as invalid, and the function should be stopped with raising an exception telling the user that the input is not acceptable. Otherwise, record the name as a field in the object for further use.

```
        if not isinstance(name, str):
            raise Exception(repr(name) + '␣is␣not␣a␣string!')
        self.name = name
```

For example, before we use a variable X in logical clauses, users need to instantiate it by writing the following code before it is used in functional terms:

```
X = Var('X')
```

In order for the users to access the string representation of Var, $\_\_repr\_\_$ should be overridden to simply return the field $name$ stored in Var.

```
    def __repr__(self):
        return self.name
```

Except for the variables that users define, there are also logical variables that generated automatically by the resolution engine. We would call these variables ”internal variables”. To make sure each of the internal variables that the engine generates differ from each other, we need a global variable var_id which recorded the current id of the internal variables. The initial value of var_id is set to zero.

```
var_id = 0
```

We define a new class $IntVar$ to represent the internal variables which is a subclass of Var.

```
class IntVar(Var):
```

In the initializer of $IntVar$, we would first use the keyword $global$ to gain access to the global variable var_id defined above, and assign the value of it to the field id in $IntVar$. The global variable var_id is then made unique by increasing the value by one. This is very important because we would like to make sure every internal variables generated by the resolution engine should have a unique name.

```
    def __init__(self):
        global var_id
        self.id = var_id
        var_id += 1
```

We also need to override the method $\_\_repr\_\_$ of $IntVar$ to give a proper string representation of the internal variables. By following the naming convention of Prolog regarding internal variables, we use the string of the value headed by $\_G$.

```
    def __repr__(self):
        return '_G' + str(self.id)
```

In order to make sure the user-defined logical variables would not have the same name with the internal variables, we need to perform check in $Var.\_\_init\_\_$ to ensure the string representation would not start with $'\_G'$.

```
class Var(object):
    def __init__(self, name):
        if not isinstance(name, str):
            raise Exception(repr(name) + ' is not a string!')
        if name.startswith('_G'):
            raise Exception('The name of variable cannot ' \
                            'start with \'_G\'')
        self.name = name
```

There is another kind of special logical variables called anonymous variable. It acts exactly as internal variables, except that it might not represent the same value in its occurrences. Because it only has one instance, first we will define a meta class of singleton. It is useful afterwards for the terms of only one instance. A meta class is a subclass of *type*, so we create a new class *Singleton* which is a subclass of *type*.

```
class Singleton(type):
```

It has a a field in it called *_instance* which would store the only instance of the singleton variable. It is initialized to None indicating no instance of the class has been created

```
    _instance = None
```

__call__ should be overridden in order to make sure that in the process of creating an instance of singleton class, there is only one instance to be created. If the field *_instance* is None, then it means there is not yet an instance created of the object, so we call the *__call__* method of its super class *type* to initialize the object and store the result to *_instance*. Otherwise, it means the instance is already created, so we do not need to initialize the object. Then, return *_instance* as the result.

```
    def __call__(cls):
        if cls._instance is None:
            cls._instance = type.__call__(cls)
        return cls._instance
```

Now, we could define a class called *AnnoVar* whose meta class is *Singleton* we defined above:

```
class AnonVar(object):
    __metaclass__ = Singleton
```

For it to be easily used by the system, we create an instance of it called _ which is the same as in Prolog.

```
_ = AnonVar()
```

Note that the class would not be used by the resolution engine. Before it is used by the engine, it will first be converted to an instance of *IntVar*. We would discuss this in detail in section 5.

5.1.2 Functional Term

For users to define functional terms in logic programming, one way is to define them in a similar way as variables:

```
term = Term('f(X)')
```

However, to make it work, we need to introduce an additional method to parse the string before it could be used by the resolution engine, which potentially slows down the program. More importantly, this would also make the program users need to write more complicated and less readable codes, because a term must be initialized explicitly.

Because we would like the users to write functional terms as in logic programming systems, and we would like a term to be initialized internally, therefore we use the idea of mixin classes so we could define the initializing method in a super class, and the initializer of subclasses just inherit the initializing method, so it could be initialized internally. We call the mixin class used to define functional terms $Term$. A functor is one subclass of $Term$, and a functional term is an instance of the class.

First, we would start by defining the mixin class $Term$. $\_\_init\_\_$ takes an ambiguous number of arguments, so we use the notation $*args$ which would be represented as a Python tuple. The functor of $Term$ is the current class $\_\_class\_\_$ which is the subclass of $Term$ defined by users. As the values might not be of the types we defined, we should use a function called $fromPythonArg$ to convert them to the types we defined. We also use lists instead of tuples, because values in tuples cannot be changed which makes applying the substitutions of the arguments more complicated.

```
class Term(object):
    def __init__(self, *args):
        self.functor = self.__class__
        self.args = [fromPythonArg(arg) for arg in args]
```

Then, users could define their own functor by creating a subclass of it that does nothing. For example, to create a functor f, users could write

```
class f(Term): pass
```

When users use a term such as $f(x, y)$, the initializing method $\_\_init\_\_$ would be called to create an instance of the mixin class $Term$. The string representation $\_\_repr\_\_$ of $Term$ need to be overridden. First, we need to convert each argument to string. Then we use the built-in string function $join$ to group the values into a string of comma-separated values. The name of the functor class can be easily obtained by using the field $\_\_name\_\_$. The result is a combination of the name of the functor, a left round bracket, the string of comma-separated arguments, and a right round bracket.

```
def __repr__(self):
        return self.functor.__name__ + '(' + \
                ','.join([repr(arg) for arg in self.args]) + ')'
```

5.1.3 Constant

As for constants, we use a similar strategy as in Prolog, that is considering them as a special case of functional term whose functor is its own value, and its argument list is empty (arity = 0).

```
class Const(Term):
    args = []
    def __init__(self, val):
        self.functor = val
```

The string representation *__repr__* of *Const* is simply the string representation of the functor.

```
    def __repr__(self):
        return repr(self.functor)
```

5.2 Unification

In this section, we will have a look at the implementation of the parts related to unification. This includes defining the data structure to store substitutions, and the implementation of Robinson's unification algorithm in the paradigm of object-oriented programming.

5.2.1 Substitution

In Python, we could use built-in dictionaries to represent the data structure where keys are the variables to be substitute, and values are the terms to substitute the variables. In Python, in order to make objects the keys of dictionaries, we must override *__hash__* and *__eq__* so that two distinct objects' hash values are not likely to equal, and use *__eq__* to define in which case two keys should be considered as equaled. Because the operators of *Var* including *__eq__* would be overridden later to provide other functionalities of the system, we need another class *Key* to wrap a variable to be used as types of keys in the dictionaries.

```
class Key(object):
```

The initializer of the object simply takes a logical variable and store it as its field.

```
    def __init__(self, var):
        self.var = var
```

Then, we override *__hash__* to simply return the hash value of the variable stored in it.

```
    def __hash__(self):
        return hash(self.var)
```

To test the equality test of one key with another object, we first need to check if another object is also a key. If so, we need to also check if the logical variable stored in both keys are the same.

```
    def __eq__(self, other):
        return isinstance(other, Key) and self.var is other.var
```

The string representation of the key would be the same as the string representation of the logical variables stored in it.

```
    def __repr__(self):
        return repr(self.var)
```

5.2.2 Robinson's Unification Algorithm

Due to the purpose of object-oriented design, instead of defining a single function that decides if two terms are unifiable and return the most general unifier, we have to define multiple functions in each class of the basic terms we defined before. Each function would have the same type of input and output.

```
def unifyWith(self, other, env):
    # returns a boolean number
    pass
```

We will also need another function called $applyEnv$ which returns the result of applying the environment to the term.

```
def applyEnv(self, env):
    # returns an instance of the class
    pass
```

Next, we will talk about the implementation of $unifyWith$ in each class of basic terms.

Variable

As mentioned in the background, a substitution is a finite set of pairs $(variable, term)$. The substitution can be represented as a series of disjoint sets in which all of the variables can be unified with some certain value. Therefore, we could use disjoint-set data structure first described by Bernard A. Galler and Michael J. Fischer in 1964 [16]. First, let us look at the three operations of disjoint-set data structure:

$MakeSet$ is the function of creating a new set consisting of the given value. This is achieved by making the value a new root in the disjoint-set forest, whose parent node is itself.

```
function MakeSet(x):
    x.parent := x
```

$Find$ is the function of finding the root node of a given node in the disjoint-set forest. This is done by recursion. If the current node is already root, then return it. Otherwise, return the root value of its parent node by recursively call $Find$ using its parent node.

```
function Find(x):
    if x.parent == x
        return x
     else
        return Find(x.parent)
```

Union is the function to union the trees of two given nodes. This could be easily done by assigning the root node of the second node to the parent node of the root node of the first node.

```
function Union(x, y):
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

In order to make the algorithm more efficient, one optimization method called ”union by rank” is applied. It means that we need to store the rank of the root. Thus, *MakeSet* need to be modified with the rank initialized to zero:

```
function MakeSet(x):
    x.parent := x
    x.rank := 0
```

Union needs to be modified as well. After finding the root nodes of the two given nodes, we check if they equal. If so, then they are already in the same tree, so there is no need to union them and the function should return. Otherwise, if the rank of the two root nodes are different, we attach the root node with smaller rank to the greater rank, so that the rank of the root node of the new union tree do not need to be modified. If the rank of the two root nodes are different, we union them in whichever order and the rank of the root node of the new union tree needs to increase by one.

```
function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     if xRoot == yRoot
         return
     if xRoot.rank < yRoot.rank
         xRoot.parent := yRoot
     else if xRoot.rank > yRoot.rank
         yRoot.parent := xRoot
     else
         yRoot.parent := xRoot
         xRoot.rank := xRoot.rank + 1
```

Another optimization method is called ”path compression”. It means that while finding the root node of a given node, we flatten the tree so that every node in the searching path is directly linked to the root node. In order to achieve this, we use recursion to redefine *Find*. If the given node is not the root node, then set the parent node of the current node to be the root node of its parent node by recursively call *Find* using its parent node. Finally, return the new parent node of the current node which is also the root node.

```
function Find(x)
    if x.parent != x
       x.parent := Find(x.parent)
    return x.parent
```

As for variables, we would implement the unification as a combination of Robinson's unification algorithm and the union-find algorithm in disjoint-set data structure. The disjoint-set forest is represented as the environment. Unlike in the original disjoint-set operations, the root of the tree can be something other than the logical variables. If the variable could be instantiated to a value, that is either a constant or a functional term, then the value should be the root of the tree, which means all the child nodes which consists of the logical variables should be instantiated with the value. Another difference is that because we are implementing the forest using Python dictionaries instead of arrays as in the original implementation. Thus, it makes more sense that the roots in the forest are simply logical variables appearing in the values of the dictionary but not appearing in the keys of the dictionary. To make a logical variable added to the set, we just need to define a new field $rank$ in Var to record the rank of the root, as in $MakeSet$:

```
class Var(object):
    rank = 0
```

The initializer of $IntVar$ also needs to be overridden to store the field $rank$.

By using the idea of union-find algorithm, $applyEnv$ is essentially trying to find the root of the disjoint set. Thus, we just implement it as the $Find$ operation.

```
class Var(object):
    def applyEnv(self, env):
        if Key(self) in env:
            env[Key(self)] = env[Key(self)].applyEnv(env)
            return env[Key(self)]
        return self
```

According to Robinson's unification algorithm, a variable can be unified with anything by just substitute it with other term. By using the idea of union-find algorithm, $unifyWith$ is essentially the $Union$ function trying to union the set of the logical variable and the set of another logical variable. If the other term is not a logical variable, then we simply substitute the logical variable with the other term.

```
    def unifyWith(self, other, env):
        self = self.applyEnv(env)
        other = other.applyEnv(env)
        if self is other:
            return True
        if isinstance(other, Var):
            if self.rank < other.rank:
                env[Key(self)] = other
            elif self.rank > other.rank:
                env[Key(other)] = self
            else:
                env[Key(self)] = other
                other.rank += 1
        else:
            env[Key(self)] = other
        return True
```

Functional Term

According to Robinson's unification algorithm, a functional term can be unified to a variable. It might also be unified with another functional term that their functor and arity are equal. If so, then we need to check if each of their arguments can be unified with each other.

```
class Term(object):
    def unifyWith(self, other, env):
        if isinstance(other, Var):
            return other.unifyWith(self, env)
        if isinstance(other, Term) and self.functor == other.functor\
           and len(self.args) == len(other.args):
            for i in range(len(self.args)):
                arg1 = self.args[i].applyEnv(env)
                arg2 = other.args[i].applyEnv(env)
                if not arg1.unifyWith(arg2, env):
                    return False
            return True
        return False
```

The implementation of *applyEnv* is trivial. First, use *copy.copy* to make a copy of itself. Then, do *applyEnv* on each of its arguments and return the functional term after the operation.

```
    def applyEnv(self, env):
        self = copy(self)
        for i in range(len(self.args)):
            self.args[i] = self.args[i].applyEnv(env)
        return self
```

5.3 Knowledge Base Creation

5.3.1 Data Structures

To store facts and rules defined by the user, we need a data structure. Because we would like to preserve the order of the user-defined facts and rules and we also want to make sure that loading facts and rules would be efficient enough, we would use an ordered dictionary, which behaves just like default dictionary with order of insertions preserved. We would use a global variable called all_rules.

```
kb = OrderedDict()
```

The keys of the dictionary should be a combination of the functor and the arity of $Term$, so when querying a specific $Term$, the system would quickly filter out the ones with different functor or arity. In Python, we could use tuples as the key:

```
key = (self.functor, len(self.args))
```

The values of the dictionary should be a list of facts and rules. In order to represent facts and rules, we need a data structure to represent the body of the clause, which could be either empty or a list of functional terms. Instead of using a list directly, we introduce a new class called $Terms$ which represents the body of the clause. The initializer of it takes a $Term$ or $Terms$ and use the initializer of its super class to initialize it. If no arguments is given, then we create an empty list.

```
class Terms(list):
    def __init__(self, term = None):
        if term:
            list.__init__(self, [term])
        else:
            list.__init__(self)
```

The *__and__* operator of $Terms$ is used to concatenate terms in the rule. It is also useful when users define rules because the body of the clause will added to a single $Terms$. If the other object is of the type $Terms$ then we just extend the other $Terms$ to the end of itself. If the other object is of the type $Term$, then we append the other $Term$ to the end of itself. Otherwise, the operation is not supported, so an exception is raised. In the end, return the modified version of itself.

```
    def __and__(self, other):
        if isinstance(other, Terms):
            self += other
        elif isinstance(other, Term):
            self.append(other)
        else:
            raise Exception(str(other) + ' is not a legal term')
        return self
```

The *__and__* operator of $Term$ is to simply wrap itself in $Terms$, and then call the the *__and__* operator of $Terms$.

```
class Term(object):
    def __and__(self, other):
        return Terms(self) & other
```

5.3.2 Negation as Failure

In Python, we use *__invert__* operator ~ to provide the feature of negation as failure. The state of invert *inverted* should be stored in $Terms$ to be used later for querying.

```
class Terms(list):
    def __init__(self, term = None):
        ...
        self.inverted = False
```

In logic programming, negation could be nested. Thus, $Terms$ should also be nested if users apply nested invert operator to it. For example, ~~$f(X)$ and $f(X)$ should not be regarded as the same. In order to enable the invert operator of $Terms$, we need to overload the *__invert__* function in $Terms$. If itself is already inverted, then in order to make it nested, we use the $Terms$ initializer to create a new $Terms$. Then, we set the field *inverted* to True and return the modified version of itself.

```
    def __invert__(self):
        if self.inverted:
            self = Terms(self)
        self.inverted = True
        return self
```

If a $Term$ is inverted, then we need to convert it to $Terms$ and then apply the invert operator.

```
class Term(object):
    def __invert__(self):
        return ~Terms(self)
```

The *__and__* operator of $Terms$ should make sure that if any of the two terms might be inverted, it has to be wrapped in a not inverted $Terms$ first before doing further operations on them.

```
class Terms(list):
    def __and__(self, other):
        if self.inverted:
            self = Terms(self)
        if isinstance(other, Terms):
            if other.inverted:
                other = Terms(other)
            ...
        ...
```

The clause would be stored in the a Python tuple $(head, body)$, where $head$ is a $Term$, and $body$ is a $Terms$. Recall that to define a fact, we use the following syntax: (H is a functional term)

```
+ H
```

Therefore, we need to define the $\_\_pos\_\_$ operator for $Term$. The body of the terms would be an empty $Terms$. First, as mentioned above, the key in the knowledge base is a tuple of functor and arity of a functional term.

```
class Term(object):
    def __pos__(self):
        key = (self.functor, len(self.args))
```

If the key could be found in the knowledge base, then we append the clause to the end. Otherwise, create a new singleton list with the clause.

```
        if key in kb:
            kb[key].append((self, Terms()))
        else:
            kb[key] = [(self, Terms())]
```

Recall that to define a rule, we use the following syntax: ($H, B_1, B_2, ...B_N$ are functional terms)

```
H <= B_1 & B_2 & ... B_N
```

Therefore, we need to define the $\_\_le\_\_$ operator for $Term$. The body of the terms would be the other side of <=. As the other side might be a singleton term, we need to wrap it in $Terms$ to make sure the type is the same overall.

```
    def __le__(self, other):
        if isinstance(other, Term):
            other = Terms(other)
        key = (self.functor, len(self.args))
        if key in kb:
            kb[key].append((self, other))
        else:
            kb[key] = [(self, other)]
```

5.4 Query

5.4.1 Backtracking

The backtracking involves with the process of doing a depth-first search on the terms we define. *State* is used to define a state in the searching process in the querying of *Terms*, in order for the engine to backtrack.
First, we need to define the initializer of *State*. It takes a *Term* or *Terms* to query for the result, an environment *env*, and another state *prev* as a reference to backtrack to the previous state.

```
class State(object):
    def __init__(self, term, env, prev):
        self.env = env
        self.prev = prev
```

We need to create a field called *gen* to generate all the possible new environments in the given environment. It is a generator object given by the result of *query* of *Term* or *Terms*. If the given term is a *Term* then we apply the environment to it before querying. If the given term is a *Terms*, then we query it passing the environment.

```
        if isinstance(term, Term):
            self.gen = term.applyEnv(env).query()
        elif isinstance(term, Terms):
            self.gen = term.query(env)
```

We provide the upper-level query a method to generate the new environment. Note that the final environment is a union of the current environment and the environment generated in the state. If there is no more environments to be generated, just return *None*.

```
    def generate(self):
        try:
            return joinEnv(self.env, next(self.gen))
        except StopIteration:
            return None
```

Union two environments is easy to implement. We need to make a copy of one environment, and call *update* to use the other environment to update it and return it.

```
def joinEnv(env1, env2):
    env1 = env1.copy()
    env1.update(env2)
    return env1
```

5.4.2 Querying $Term$

Querying a $Term$ consists of following steps:

1. Try to find the rule in the dictionary.

   ```
   class Term(object):
       def query(self):
           key = (self.functor, len(self.args))
           if key not in kb:
               return
           rules = kb[key]
   ```

2. For each found rule:

   ```
           for left, rights in rules:
   ```

   (a) int_names is a dictionary which records the mapping from variables in the term to the unique variables generated internally. Make the variables in the head of the clause unique by calling the method $unique$.

   ```
               int_names = {}
               left = self.unique(int_names, left)
   ```

   (b) Make the variables in the body of the clause unique by calling the method $unique$ for each of its element if the body is not empty.

   ```
               if rights:
                   rights_new = Terms()
                   rights_new.inverted = rights.inverted
                   rights_new += [self.unique(int_names, right)
                                  for right in rights]
                   rights = rights_new
   ```

   (c) Test if the rule can unify with the head of the rule.

   ```
               if not self.unifyWith(left, env):
                   continue
   ```

   (d) Yields the result of querying the body of the rule.

   ```
               for res in rights.query(env):
                   yield res
   ```

5.4.3 Querying $Terms$

Querying a $Terms$ involves with the implementation of backtracking mechanism.

1. First, we would start with some essential variables. $index$ is used to record the current index in the list. $prev$ is used to record the previous state of the current state.

```
class Terms(list):
    def query(self):
        index = 0
        prev = None
```

2. While the index non-negative:

```
        while index >= 0:
```

   (a) If $Terms$ reach the end of the body, which means the query of all the terms in it succeeds,

```
            if index == len(self):
```

      i. If the $Terms$ is inverted, then it means the query of the $Terms$ fails, so the function just return with no values.

```
                if self.inverted:
                    return
```

      ii. Otherwise, yields the result and continue with the query supposing the last term fails. Set the state to $prev$ and decrease $index$ by one.

```
                yield env
                state = prev
                index -= 1
```

   (b) Otherwise the current state should be a new state, so call the initializer of $State$ to create a new state.

```
            else:
                state = State(self[index], env, prev)
```

   (c) Generates a new environment of the state

```
            env = state.generate()
```

(d) While the environment generated is $None$ which means query of the current state fails, backtrack to the previous state iteratively until discovered a state that generates an environment, so the querying process can continue.

```
while env is None:
```

i. While the generated environment is None, we go to the previous state and update $index$.

```
state = state.prev
index -= 1
```

ii. If it backtracks to a state of None, then the backtracking/querying process should be ended. If query of the all the terms in $self$ fails, then the query of the $Terms$ succeeds with an empty environment if it is inverted. If the backtracking succeeded, then we use the state to generate a new environment and iteratively repeat the backtracking.

```
if state is None:
    if self.inverted:
        yield {}
    return
env = state.generate()
```

(e) Update $index$ and $prev$ and try again.

```
index += 1
prev = state
```

5.5 Conversions Between Terms

5.5.1 From Python Terms to Logical Terms

To convert from Python terms to the logical terms in `LogicAPI`, there are four cases to be think of:

```
def fromPythonArg(arg):
```

1. If the term is an anonymous variable, then we convert it to an internal variable.

```
if isinstance(arg, AnonVar):
    return IntVar()
```

2. If the term is a logical variable, we reset the rank of it to zero and return it.

```
elif isinstance(arg, Var):
    arg.rank = 0
    return arg
```

3. If the term is already a $Term$ in the system, just return it.

```
    elif isinstance(arg, Term):
        return arg
```

4. Otherwise, we check if its type is a supported constant type. We define a set called $supportedConstTypes$ which include all the types valid for logical constants. If the type of the term is a valid constant type, then we wrap it in $Const$ class we defined and return it. Otherwise, raise an exception to tell the user that the type is not supported.

```
    else:
        for t in supportedConstTypes:
            if isinstance(arg, t):
                return Const(arg)
        raise Exception("Unsupported constant type of " +
                        repr(arg))
```

5.5.2 From Logical Terms to Python Terms

Converting from logical terms to Python terms are trivial, as at the time, the system could only send values of logical constants. If it is a constant, then set it to the $functor$ of it which is the original value of the constant in Python. Otherwise, just return the term untouched.

```
def toPythonArg(arg):
    if isinstance(arg, Const):
        arg = arg.functor
    return arg
```

5.6 Processing Output

In this section, we would talk about how to process the output of the system. This includes the representation of the result and the means we provide for users to get the result.

5.6.1 Representing result

To properly represent the result, we create a new class called $Result$ which is a simplified version of the Python dictionary. Users can only see the representation and get items in it. It takes an ordered dictionary and stored it.

```
class Result(object):
    def __init__(self, data):
        self.data = data
```

The string representation is similar to the one of Python dictionary, except that we use = to represent the mapping between keys and values.

```
    def __str__(self):
        return '{' + ', '.join(
                [repr(k) + ' = ' + repr(v)
                for k, v in self.data.items()]) + '}'
```

To get the value of a variable, we need to convert it to the built-in Python data types, so we called the function *toPythonArg* defined before.

```
    def __getitem__(self, var):
        return toPythonArg(self.data[Key(var)])
```

We need also an interface for users to check whether a variable is in the result or not

```
    def __contains__(self, var):
        return Key(var) in self.data
```

Finally, we define a global function for users to query a term, and returns a generator object of *Result* we defined above. In order to make sure everything is ordered, and only the visible variable will be recorded in the final result. We define a function called *variables_list* to record the list of all the visible variables, and use it to reorder the original result.

```
def query(x):
    for env in x.query():
        l = []
        variables_list(x, l)
        res = OrderedDict()
        for var in l:
            if var not in res:
                res[var] = var.applyEnv(env)
```

Before outputting the result, in order to make sure we only show the equaled relation between the user-defined variables, instead of mapping them to the same internal variable as in the original result, we have to group them using a reversed dictionary, and then use it to remap the result.

```
        rev = defaultdict(list)
        for var in res:
            if isinstance(res[var], Var):
                rev[res[var]].append(var)
        for l in rev.values():
            for i in range(1, len(l)):
                res[l[i - 1]] = l[i]
            del res[l[len(l) - 1]]
        yield Result(res)
```

5.7 Lists

In this section, we would talk about the implementation of lists. In order to make it efficient, and able to be pattern matched. The implementation would not use the Python lists whose underlying structure is array. It would not follow the implementation of lists in Prolog which is a functional term, and could be regarded as a singly linked list. We use doubly linked lists with node sharing instead for better control of the structure which should gives a better performance.

5.7.1 Empty List

To define lists, we first need to define the empty list which is nothing but a singleton constant.

```
class EmptyList(Const):
    __metaclass__ = Singleton
```

The initializer sets the functor to the class of $EmptyList$.

```
    def __init__(self):
        self.functor = EmptyList
```

Next subsection includes the implementation of the class $List$. Unlike other special terms, $List$ is a little more complicated. We would talk about the implementation ideas and other related operations.

5.7.2 Non-Empty Lists

Recall that lists in Prolog are a special functional term whose functor is . and arguments are its head and tail. Thus, we can make $List$ a subclass of $Term$, and the implementation is trivial.

However, if we implement it this way, it would be essentially a singly linked list, using which, changing the last elements would be very time-consuming.

Therefore, instead of making use of the $Term$, we create a completely new class $List$ which essentially a doubly linked list. We start with the implementation of $Node$ which is used to store the actual value in lists and two pointers to the neighbor nodes.

```
class Node(object):
    def __init__(self, val, next = None, prev = None):
        self.val = val
        self.next = next
        self.prev = prev
```

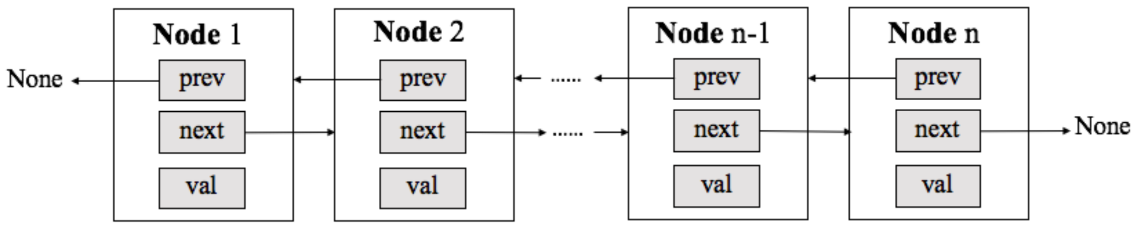


Figure 5.1: Structure of *Node*s forming a doubly linked list

To make the initialization of lists simpler, we also introduce a method called *add* to wrap up a value in *Node* and put it after the node.

```
    def add(self, val):
        self.next = Node(val, self)
        return self.next
```

After this, we could implement the initializer of *List*, which takes a python list and possible unknown rest of the lists. The unknown rest part of lists could either be *None* or an instance of *Var*. It is useful for pattern matching lists, and its usage will be discussed later.

```
class List(object):
    def __init__(self, l, rest = EmptyList()):
        self.first = Node(fromPythonArg(l[0]))
        temp = self.first
        for i in xrange(1, len(l)):
            temp = temp.add(fromPythonArg(l[i]))
        self.last = temp
        self.rest = rest
```

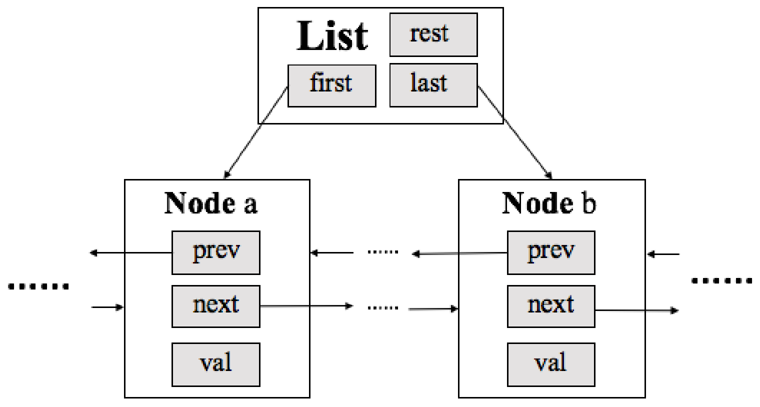


Figure 5.2: Structure of a *List*

__add__ operator is used to concatenate two *List*s. To concatenate two *List*s, first we need to concatenate the underlying list of nodes. We need to modify the pointer of *next* of the last node in the first *List* to make it point to the first node in the second *List*. We also need to modify the pointer of *prev* of the first *List* to make it point to the last node in the first *List*. Then we make a new *List* whose field *first* is the same as the field *first* of the first *List* and the field *last* is the same as the field *last* of the second *List*.

```
def __add__(self, other):
    if not isinstance(other, List):
        raise Exception(repr(self) + '␣and␣' + repr(other) +
                        '␣cannot␣be␣concatenated.')
    if self.rest:
        raise Exception(repr(self.rest) + '␣in␣' +
                        repr(self) + '␣is␣unknown.')
    self.last.next = other.first
    other.first.prev = self.last
    other = copy(other)
    other.first = self.first
    return other
```

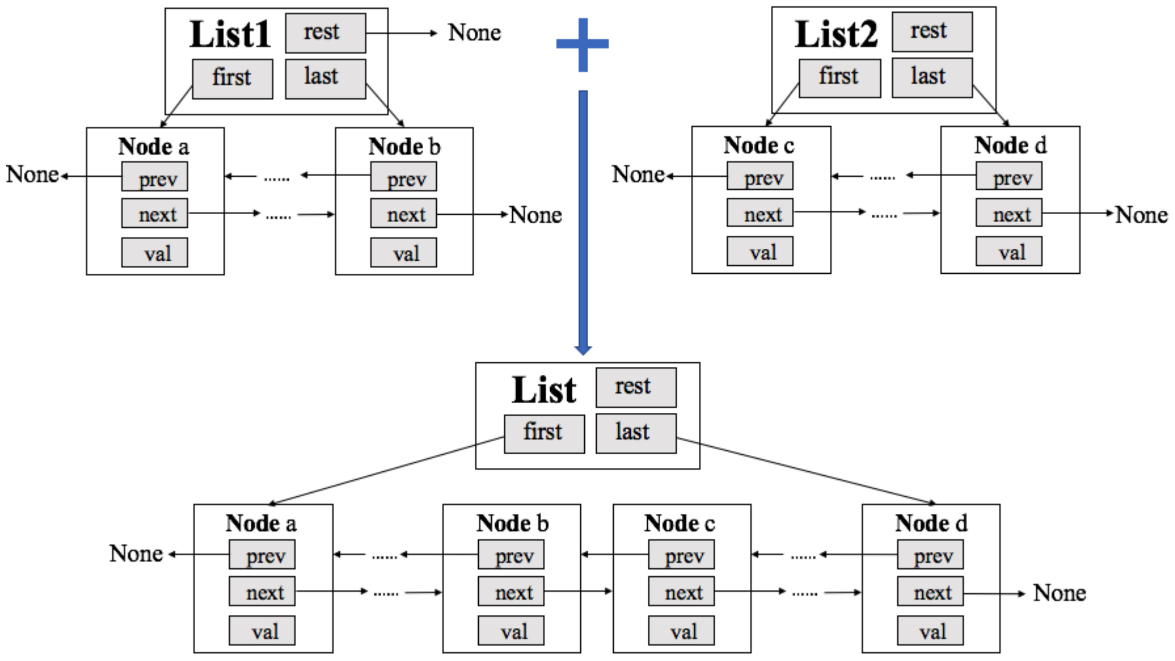


Figure 5.3: Concatenation of two *List*s

Apply substitutions to a list is trivial, as it just applies substitutions to every elements in the list. It could be easily implemented as following:

```
def applyEnv(self, env):
    self = deepcopy(self)
    temp = self.first
    while temp:
            temp.val = temp.val.applyEnv(env)
        if temp is self.last:
            break
        temp = temp.next
```

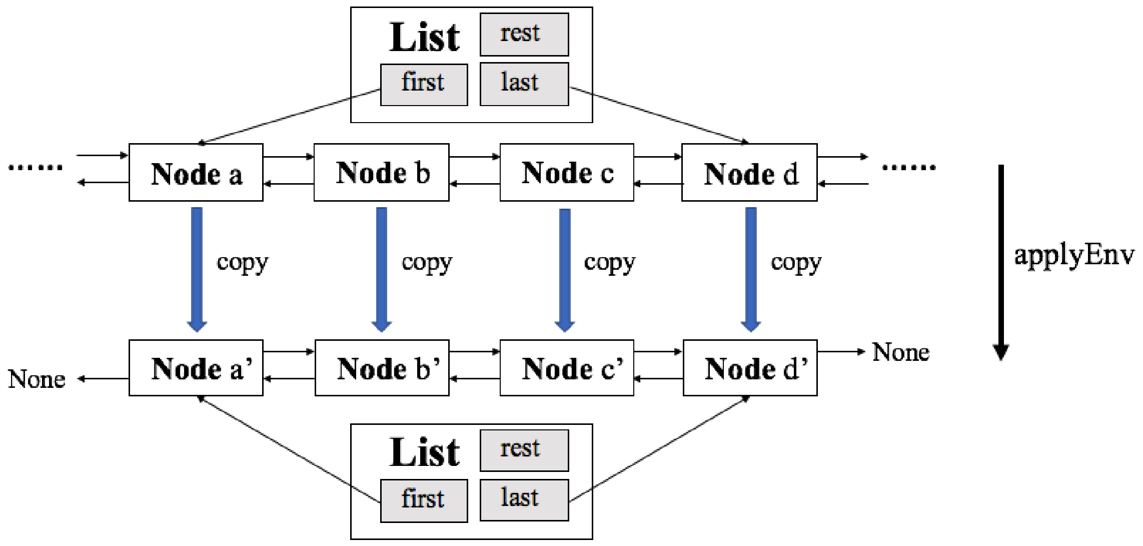


Figure 5.4: Applying *applyEnv* to a *List*

However, using *deepcopy* could make the program very inefficient. As it has to copy every information in the nodes, and might copy the nodes beyond the range of the list if there are elements before $self.first$ or after $self.last$. A better implementation would be building a new list using $self$, and call *applyEnv* while putting the elements into the new nodes.

```
def applyEnv(self, env):
    first = Node(self.first.val.applyEnv(env))
    temp = first
    p = self.first.next
    while p:
        temp = temp.add(p.val.applyEnv(env))
        if p is self.last:
            break
        p = p.next
    self = copy(self)
    self.first = first
    self.last = temp
    self.rest = self.rest.applyEnv(env)
```

Before we output $self$, remember that $self.rest$ might unify to another list, in which case we had to concatenate it to $self$ before outputting.

```
        if isinstance(self.rest, List):
            self.last.next = self.rest.first
            self.rest.first.prev = self.last
            self.last = self.rest.last
            self.rest = self.rest.rest
        return self
```

To unify a list with another a list, we try to unify each of them one by one. Unlike in Prolog where lists are functional terms. In our system, list is a wrapper for nodes, so when the comparison came to the end of any of the ends of the two lists, there are three base cases to be think of:

1. The first list reaches the end while the second does not.
2. The second list reaches the end while the first does not.
3. The two lists reach the end at the same time.

```
    def unifyWith(self, other, env):
        if isinstance(other, Var):
            return other.unifyWith(self, env)
        if isinstance(other, List):
            self = copy(self)
            other = copy(other)
            temp1 = self.first
            temp2 = other.first
            while temp1 and temp2:
                if not temp1.val.unifyWith(temp2.val, env):
                    return False
                if temp1 is self.last:
                    temp1 = None
                if temp2 is other.last:
                    temp2 = None
                if temp1:
                    temp1 = temp1.next
                if temp2:
                    temp2 = temp2.next
            if temp1:
                self.first = temp1
                return other.rest and other.rest.unifyWith(self, env)
            if temp2:
                other.first = temp2
                return self.rest and self.rest.unifyWith(other, env)
            return self.rest.unifyWith(other.rest, env)
        return False
```

5.8 Other Special Terms

5.8.1 Functions

In order to use the Python functions in logic programming, we introduce the *Func* class as a special case of functional term. Unlike general functional terms, it needs an additional function argument to pass the function object. The reason for this is that the name of the function has already been occupied. It would cause the function overloaded, which means the functionality of the original function is lost. Therefore, we have to pre-store the function somewhere, or pass it as an argument like the definition below.

```
class Func(Term):
    def __init__(self, func, *args):
        self.functor = func
        self.args = [fromPythonArg(arg) for arg in args]
```

Firstly, we need a new function for *Func* to evaluate itself and returns the output.

```
    def eval(self):
        for i in range(len(self.args)):
            if isinstance(self.args[i], Var):
                raise Exception('There are varibles in the function')
            elif isinstance(self.args[i], Func):
                self.args[i] = self.args[i].eval()
            elif isinstance(self.args[i], Const):
                self.args[i] = self.args[i].functor
        return self.functor(*self.args)
```

Each time the function is used in unification or querying, it will evaluate itself.

```
    def unifyWith(self, another, env):
        return another.unifyWith(fromPythonArg(self.eval()), env)
    def query(self):
        self.eval()
        yield {}
```

By using *Func*, one thing that we could implement the arithmetic operations in `LogicAPI`. Take the addition operation $+$ as an example, we first define a logical function *Add* which is a subclass of *Func*:

```
class Add(Func):
```

The function is just the same as the built-in operator. We could access the function of the built-in operator by using the one in the library *operator*, so we can simply assign the function to the field $function$.

```
    function = operator.add
```

To enable the operator on logical variables, we need to override the function *__add__* and *__radd__* of Var to wrap the given arguments in *Add*:

```
class Var(object):
    def __add__(self, other):
        return Add(self, other)
    def __radd__(self, other):
        if isinstance(other, list):
            return List(other, self)
        return Add(other, self)
```

Therefore, we could have functional term like $sum3/4$ which computes the sum of three values defined as follows:

```
+sum3(X, Y, Z, X + Y + Z)
```

In order to make addition of logical functions possible, we also need to override the function *__add__* and *__radd__* of $Func$:

```
class Func(Term):
    def __add__(self, other):
        return Add(self, other)
    def __radd__(self, other):
        return Add(other, self)
```

Other similar operators like $-$, $*$, / could be defined in the same way. If you would like the result of the function considered by the resolution engine while backtracking, it is nice to have a particular type of function whose boolean value of output would be considered as success or failure.

```
class BoolFunc(Func):
    def query(self):
        if self.eval():
            yield {}
```

It is useful to define operators like ”greater than”. The definition is similar to defining ”add” operator, except that it is a subclass of $BoolFunc$ instead of $Func$.

```
class GT(BoolFunc):
    function = operator.gt
```

5.8.2 Cut

Cut is a special term that tells the engine not to backtracking on it. The definition is trivial.

```
class Cut(Term):
    __metaclass__ = Singleton
```

We have to introduce another variable in $Terms$ to make if it is cut, so that the upper level of the query tree would know whether it should search the rest of the tree.

```
class Terms(list):
    def __init__(self, term = None):
        ...
        self.cut = False
```

A *Terms* would be set to 'cut' if it queries the special term *Cut*. Because it should not be backtracked, we should also set the start of the querying index to be the next one in *Terms*, so when it tries to backtrack to the index of *Cut*, the querying process would be directly stopped.

```
    def query(self):
        ...
        start = 0
        while index >= start:
            ...
            if isinstance(self[index], Cut):
                index += 1
                start = index
                self.cut = True
                continue
            ...
```

While querying of the *Term*, we need to check if the right-hand side *Terms* is cut. If so, then we need to stop the querying process as well.

```
class Term(object):
    def query(self):
        ...
        for left, rights in rules:
            ...
            if rights.cut:
                return
```

5.9 Object-Oriented Paradigm

To enable the object-oriented paradigm in the system, we could see objects in the logic programming as logical constants. Therefore, we define a new class *Object* which just a subclass of *Const*:

```
class Object(Const): pass
```

However, for a logical constant to be used in the system, *functor* must be defined. Although we could make users to define the field by themselves, this is not really a good method. Therefore, we again use the idea of metaclass to modify the behavior of the initializer defined by users, so the field *functor* could be added internally. we define a new class called *ObjectType* which is a metaclass. We overload the initializer, so the field *functor* which is just a reference to the object would be added to classes using it as a metaclass while initialization. In object-oriented paradigm, The objects can be distinguished by the reference to the object.

```
class ObjectType(type):
    def __init__(self, *args, **kwargs):
        type.__init__(self, *args, **kwargs)
```

The objects can be distinguished by the reference to the object.

```
        self.functor = self
```

Now, the definition of *Object* would be

```
class Object(Const):
    __metaclass__ = ObjectType
```

The users can make a subclass of *Object* to enable the object-oriented paradigm, and define a class of functor to be used inside the class. For example, to record the age of a person and make queries related to age, we could define the class *Person* as below:

```
class Person(Object):
    class age(Term): pass
    def __init__(self, name, age):
        self.name = name
        +self.age(self, age)
    def __repr__(self):
        return self.name
```

To ask for the persons with age smaller than 18, we could make queries like this:

```
query(Person.age(P, A) & (A < 18))
```

Some further optimization techniques are applied to the system, but would not be discussed here in detail. The main part is to provide a faster way to copy the objects without the possibility that the original object would be modified.

Chapter 6

Result

In this chapter, we will have a look at the implementation of the solver of river-crossing puzzle in `LogicAPI`, and how it could be used to visualize the solutions.

6.1 River-Crossing Puzzle Solver

The solutions of the river-crossing puzzle could now be computed in Python with the help of `LogicAPI`.

For example, if we need to define *visited*/2 to see if a state is present in the sequence of visited states. The equality is given by the predicate *equiv* already. First we need to create the logical variables that we need.

```
St1 = Var('St1')
St2 = Var('St2')
```

Then, we need to create the functor *visited*.

```
class visited(Term): pass
```

Sometimes, in the new system, it is better to use Python functions in the logical clauses to make the code more readable and more efficient.

For example, in Prolog, to compare whether two states which are lists of farmers, wolves, cabbages and goats, one method is to count the numbers of the objects and then compare the numbers.

```
count([], 0, 0, 0, 0).
count([f | Bank] , F, W, G, C) :-
        count(Bank, OldF, W, G, C),
        F is OldF + 1.
count([w | Bank] , F, W, G, C) :-
        count(Bank, F, OldW, G, C),
       W is OldW + 1.
count([g | Bank] , F, W, G, C) :-
        count(Bank, F, W, OldG, C),
       G is OldG + 1.
```

```
count([c | Bank] , F, W, G, C) :-
        count(Bank, F, W, G, OldC),
        C is OldC + 1.
```

Of course, by using accumulators, this program could be more efficient with the help of the last call optimization, but that would make the code more complex and unreadable. In `LogicAPI`, we could make use of Python functions in the case:

```
class count(Func):
    def function(self, l):
        f = 0
        g = 0
        c = 0
        w = 0
        for i in l:
            if i == 'f':
                f += 1
            if i == 'g':
                g += 1
            if i == 'c':
                c += 1
            if i == 'w':
                w += 1
        return [f, g, c, w]
```

Then the predicate $equiv/2$ could be defined using this $Func$:

```
equiv([N1, S1], [N2, S2]) <=
    (count(N1) == count(N2)) & \
    (count(S1) == count(S2))
```

For more details of the definitions of the logical clauses, please see the appendix C.

6.2 Visualization

In order to use Python libraries to visualize the result, we need to pose a query about the solutions of the problem first. This could be done by:

```
q = query(succeeds(Seq))
```

The return value is a generator object. If we need to just get one solution, then use $next$ to print the next solution.

```
next(q)
```

If a $StopIteration$ exception is raised, then it means that there are no more solutions to be generated. We could also use $list$ to convert the generator to list, so we could obtain all the solutions to the query, and have a look at all the solutions.

```
print list(query(succeeds(Seq)))
```

The printed result is as follows:

```
[{Seq = [[['f','g','w','c'],[]],
         [['w','c'],['f','g']],
         [['f','w','c'],['g']],
         [['c'],['f','w','g']],
         [['f','g','c'],['w']],
         [['g'],['f','c','w']],
         [['f','g'],['c','w']],
         [[],['f','g','c','w']]]},
 {Seq = [[['f','g','w','c'],[]],
         [['w','c'],['f','g']],
         [['f','w','c'],['g']],
         [['w'],['f','c','g']],
         [['f','g','w'],['c']],
         [['g'],['f','w','c']],
         [['f','g'],['w','c']],
         [[],['f','g','w','c']]]}]
```

In Python we could obtain the result by using the following code:

```
for result in query(succeeds(Seq)):
    solution = result[Seq]
    ...
```

By using the graphics library *pygame*, we could visualize the results sent from *LogicAPI*. The result is a program auto-solving the river crossing puzzle and visualize it. For more details about the implementation, please see the appendix D.

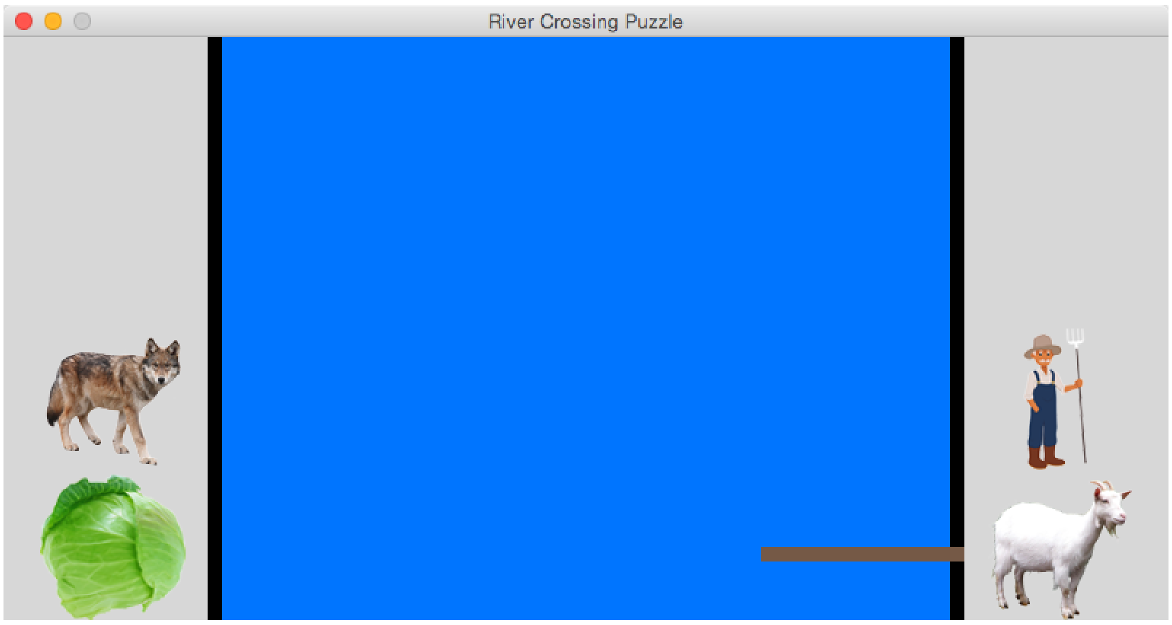


Figure 6.1: Screenshot of the program execution

Chapter 7

Evaluation

In this chapter, we evaluate the performance of the system by using the puzzle solver we implemented in `LogicAPI`. We would also compare the definitions of clauses defined purely in the system, and definitions with the aid of Python functions.

7.1 River-Crossing Puzzle

We could use the river-crossing puzzle solver in `LogicAPI` to evaluate the performance of the system while dealing with a problem. Note that it is only testing the time of querying the term for the result. The visualization part is purely implemented in Python, therefore it is not considered here.

To test the time of running the program, we could use the following codes:

```
def test():
    for i in xrange(1000):
        list(query(succeeds(Seq)))

import time
start_time = time.time()
test()
print("--- %s seconds ---" % (time.time() - start_time))
```

The output of the program in shown as below:

```
--- 121.635540009 seconds ---
```

Thus, the average time for solving the problem using *LogicAPI* is about 0.12 seconds. We can then test the memory usage in 0.1 seconds by using the following codes:

```
from memory_profiler import memory_usage
mem_usage = memory_usage(test)
print('Maximum memory usage in .1 seconds: %s MB' % max(mem_usage))
```

The output of the program is shown as below:

```
Maximum memory usage: 14.05078125 MB
```

As pyDatalog seems to be the only other system using which we could implement the same puzzle solver, We implemented it, and the implementation is roughly the same without cut operator. (See appendix B) We use the same codes to test the time and memory usage of solving the problem in pyDatalog. The output of the program is shown as below:

```
--- 980.73489666 seconds ---
Maximum memory usage: 24.40625 MB
```

Thus, we could see that the time usage of `LogicAPI` is about 12% of the time usage of pyDatalog. The memory usage of `LogicAPI` is about 57% of the memory usage of pyDatalog.

I think the main reason that it takes such long time for pyDatalog to compute the solution is that the implementation depends very heavily on the list operations. In pyDatalog, lists are represented as Python tuples so that they are immutable. This saves the time of the conversions between Python terms and logic terms as in `LogicAPI`, but during the recursions over the list, there are too many operations to copy the tuple each time it might be modified.

7.2 Feature Comparison

In the table below, we compare some special features in Prolog, pyDatalog and the new system `LogicAPI`:

Feature	Prolog	pyDatalog	`LogicAPI`
Cut	YES	NO	YES
Python Functions	NO	YES	YES
Logic Functions	NO	YES	NO
OO Paradigm	NO	YES	YES
Anonymous Variables	YES	NO	YES
Nested Negations	YES	NO	YES
Complex Terms	YES	NO	YES
Pattern Matching for Lists	YES	YES (length must be known)	YES

7.3 Python Functions vs Determinate Clauses

In this section, let us first look at three possible implementation of $reverse/2$ in `LogicAPI`. A naive definition is given as below:

```
+reverse([], [])
reverse([H]+T, L) <= reverse(T, L2) & (L == L2+[H])
```

A version with accumulator is given as below:

```
reverse(L, L2) <= reverse(L, L2, [])
+reverse([], L, L)
```

```
reverse([H]+T, L, L2) <= reverse(T, L, [H]+L2)
```

A version with the help of Python function is given as below:

```
class pyReverse(Func):
    def function(self, l):
        return l[::-1]
+reverse(X, pyReverse(X))
```

As tested, the performances of the first two definitions are similar, and the naive one is a bit faster than the one with accumulator and takes less space. This is very different as in Prolog. One reason for this is that the append operator $+$ only takes $O(1)$ time. There is also no last call optimization done for the system, so the one with accumulator would have no advantage over the naive definition.

The third definition gives the best performance. Although it takes time to convert between $List$ and list in python, it avoids the backtracking mechanism, which takes much time and space. Thus, in `LogicAPI`, this is the preferred way to define predicates involving determinate computations.

Chapter 8

Future Work

In this chapter, we would discuss how the current system might be improved in the future by adding features or optimizing it to improve the performance of the system.

8.1 More Features

8.1.1 Built-in Predicates

In order to test the correctness of the system, some built-in predicates available in Prolog were implemented. After this, we could test whether they gave the same results, in the same order. The implemented built-in predicates include $append/3$ and $member/2$. There are many more predicates that are not yet defined. It is a good idea to define more built-in predicates and ensure the best performance because we have the access to the underlying structure. Predicates like $findall/3$ are very important in logic programming, but currently it is not defined. Without understanding and accessing the underlying structure, it is not yet possible for users to define it directly.

8.1.2 Or Operator

Now, to define the body of a logical clause, we could only use and operator '&' in `LogicAPI` like ',' in Prolog. The terms form a $Terms$ by using the and operator. However, there are no or operator '—' like ';' in Prolog. Although all the clauses can be written by using and operator only, because we can split a clause with or operator to multiple clauses, there are cases where when using or operator, the style and readability can be improved. For example, in Prolog, we could do the following:

```
a :- b, (c; d)
```

While in the system now, the alternative codes you can write the identical definition by using cuts, but we need to introduce another predicate:

```
a() <= b() & temp()
temp() <= c()
temp() <= d()
```

8.1.3 Chain of Else-If-Clauses

In Prolog, there is a very interesting syntax called the chain of else-if-clause. It can be written as below:

```
a :- ( b ->
         c
     ; d ->
         e
     ; f ).
```

It means a succeeds if the following succeeds:

1. If b succeeds, check if c succeeds.
2. Else if d succeeds, check if e succeeds.
3. Else check if f succeeds

This syntax is useful. For example, if we are to define a predicate which finds the maximum value in two values, by using cut operator now, we can define it as follows

```
max(A, B, A) <= (A >= B) & Cut()
+max(A, B, B)
```

However, by using the else-if-clause chain, it is more readable. In Prolog, we could write the predicate as follows

```
max(A, B, C) <= (   A >= B
                 -> C = A
                 ;  C = B)
```

8.1.4 Updating Knowledge Bases as Dynamic Databases

Now the system allows users to add logical clauses including facts and rules to the knowledge base. However, we could not dynamically update the knowledge base as a dynamic database. For example, we might like to add clauses or delete clauses in the querying process. For example, to create a new predicate $result/1$ to just store the values succeeded in querying in another predicate. In Prolog, we could write like this:

```
succeeds(X) :-
    ...,
    assertz(result(X)).
```

The knowledge base would be dynamic during the querying process.

8.2 More Optimizations

8.2.1 Recursion to Iteration

In the current version of the system, the implementation of resolution engine depends heavily on mutual recursion. For example, perform $query$ of a $Term$ would call $query$ of a $Terms$, which would build $State$s and generate the result by using $query$ of a $Term$ or $Terms$. In this way, it is more easy to understand and build a system in a rather naive way. However, when using mutual recursions, there would be more space used to store the function calls temporarily, and it would also takes more time in order to do it. In the implementation of Prolog, the strategy that is normally used is a control stack which consists of goals to be solved. This enables the control system working using iterations instead of recursions, so the performance of the system could be improved.

8.2.2 Last Call Optimization

Once we modify the implementation of resolution by using control stack, we could further improve the space usage of the system by applying last call optimization. For example, let us look at the following example:

```
for(Int, Int, _).
for(Int, Lower, Upper) :-
        Lower < Upper,
        Next is Lower + 1,
        for(Int, Next, Upper).
```

In the second clause, the first two terms in the body are determinate, so there is no need to backtrack when it comes to the third term. Thus, we could just return the result of querying the third term as the result, and there is no need to store the previous goal in the goal block. [27]

Chapter 9

Conclusion

Overall, to complete this project, the analysis of the previously existing systems was first made to decide what their advantages and disadvantages are. By using the analysis combined with the requirements, the syntax and the structure of the system were designed. Then, the library was implemented, and multiple techniques such as design patterns, tests and optimization methods were used to ensure the functionality and the performance of the system. A visualization of the river-crossing puzzle has been implemented using the completed library and the performance of the puzzle solver was evaluated. Now, we have a library with the basic functionalities, but could be further improved or extended in future.

During the period of the project, there were times where the wrong decisions were made in the beginning, so refactoring the system were needed. For example, in the first, the unification algorithm is a function that tries to unify a term with another according to their types. This is the same as the original Robinson's unification algorithm, but in the later, I found that each time I introduced a new type, I had to add more cases to the function, which made the function very long and difficult to be understood. As we use the object-oriented paradigm, I thought making the unification method a class method in each term would be a better idea. After refactoring, the system became clearer to be understood.

I have learned much from this project. While I am working on the implementation of the unification algorithm and the resolution strategy, I began to have a better understanding about how logic programming actually works and it is not that simple as it seems. It is also a good chance for me to be more familiar with Python and its great features. As I did not have much experience coding in Python, I am learning Python the same time as developing the library.

As mentioned in chapter 8, there is still future work to do to improve or extend the system. It would be available at: `https://github.com/ZimuQin/LogicAPI`

Bibliography

[1] About python. https://www.python.org/about/. pages 7

[2] pyDatalog. https://sites.google.com/site/pydatalog/home. pages 14

[3] Python.org. https://www.python.org/. pages 6

[4] Resolution in prolog. http://athena.ecs.csus.edu/~logicp/unification-resolution.html. pages 6

[5] Standard operators as functions. https://docs.python.org/2/library/operator.html. pages 8

[6] Puzzles in education - an old classic problem. http://www.puzzles.com/PuzzlesInEducation/HandsOnPuzzles/OldClassicProblem.htm, Dec 2007. pages 10

[7] Python dictionary. https://www.tutorialspoint.com/python/python_dictionary.htm, Aug 2017. pages 7

[8] Python lists. https://www.tutorialspoint.com/python/python_lists.htm, Aug 2017. pages 7

[9] ANGELETTI, G. Understanding python's underlying data structures. http://blackecho.github.io/blog/programming/2016/03/23/python-underlying-data-structures.html, Mar 2016. pages 7, 8

[10] BOIZUMAULT, P. *The implementation of Prolog*. Princeton University Press, 2014. pages 4, 5

[11] BRATKO, I. *Prolog programming for artificial intelligence*. Pearson education, 2001. pages 1

[12] BRITTON, J. The wolf, the goat, and the cabbage. http://britton.disted.camosun.bc.ca/jbwolfgoat.htm. pages 10

[13] CERI, S., GOTTLOB, G., AND TANCA, L. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering 1*, 1 (1989), 146166. pages 14

[14] CLOCKSIN, W., AND MELLISH, C. S. *Programming in PROLOG*. Springer Science & Business Media, 2003. pages 6

[15] DELORD, C. Pylog. `http://cdsoft.fr/pylog/index.html`, Jul 2009. pages 12

[16] GALLER, B. A., AND FISHER, M. J. An improved equivalence algorithm. *Communications of the ACM 7*, 5 (Jan 1964), 301303. pages 23

[17] HARTLEY, J. Time-complexity. `https://wiki.python.org/moin/TimeComplexity`, Jun 2017. pages 7

[18] HOGGER, C. J. Essentials of logic programming. pages 6

[19] LLOYD, J. W. Practical advtanages of declarative programming. In *GULP-PRODE (1)* (1994), pp. 18–30. pages 1

[20] LLOYD, J. W. *Foundations of logic programming*. Springer Science & Business Media, 2012. pages 3

[21] MEYERS, C. Prolog in python. introduction. `http://www.openbookproject.net/py4fun/prolog/intro.html`. pages 6, 11

[22] PETERSON, I. Tricky crossings. `https://www.sciencenews.org/article/tricky-crossings`, Sep 2013. pages 10

[23] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM) 12*, 1 (1965), 23–41. pages 4

[24] SUMMERFIELD, M. *Rapid gui programming with python and qt: the definitive guide to pyqt programming*. Prentice Hall, 2015. pages 6

[25] TEKOL, Y. pyswip 0.2.2. `https://pypi.python.org/pypi/pyswip`. pages 13

[26] VAN ROSSUM, G., ET AL. Python programming language. In *USENIX Annual Technical Conference* (2007), vol. 41, p. 36. pages 1

[27] WARREN, D. H. Optimizing tail recursion in prolog. *Logic Programming and its Applications* (1986), 77–90. pages 53

Appendix A

Puzzle Solver in Prolog

```
safe(Bank) :-
        \+ (
                \+ member(f, Bank),
                member(g, Bank),
                \+ (
                        \+ member(c, Bank),
                        \+ member(w, Bank)
                )
        ).

goal([]-_).

count([], 0, 0, 0, 0).
count([f | Bank] , F, W, G, C) :-
        count(Bank, OldF, W, G, C),
        F is OldF + 1.
count([w | Bank] , F, W, G, C) :-
        count(Bank, F, OldW, G, C),
        W is OldW + 1.
count([g | Bank] , F, W, G, C) :-
        count(Bank, F, W, OldG, C),
        G is OldG + 1.
count([c | Bank] , F, W, G, C) :-
        count(Bank, F, W, G, OldC),
        C is OldC + 1.

equiv(N1-S1, N2-S2) :-
        count(N1, FN, WN, GN, CN),
        count(N2, FN, WN, GN, CN),
        count(S1, FS, WS, GS, CS),
        count(S2, FS, WS, GS, CS).

visited(S1, [S2 | _])   :- equiv(S1, S2), !.
visited(S1, [_  | Seq]) :- visited(S1, Seq).


remove(X, [X | Tail], Tail).
remove(X, [Head | Tail], [Head | NewTail]) :- remove(X, Tail, NewTail).
```

```
chooseWithResult([f | Items], Bank, NewBank) :-
        remove(f, Bank, TempBank),
        chooseRestWithResult(Items, TempBank, NewBank).

chooseRestWithResult([], Bank, Bank) :-
        safe(Bank).
chooseRestWithResult([X], Bank, NewBank) :-
        remove(X, Bank, NewBank),
        safe(NewBank).


journey(N1-S1, N2-S2, Sts, [[N2, Items, S1], [N1, [], S1] | Sts]) :-
        chooseWithResult(Items, N1, N2),
        append(Items, S1, S2).

journey(N1-S1, N2-S2, Sts, [[N1, Items, S2], [N1, [], S1] | Sts]) :-
        chooseWithResult(Items, S1, S2),
        append(Items, N1, N2).


succeeds(Sequence) :-
        extend([[f,w,g,c]-[]], Sequence, []).

extend([N-S | VisitedSeq], Seq, _) :-
        goal(N-S),
        reverse([N-S | VisitedSeq], [], Seq).
extend([LastState | VisitedSeq], Seq, Sts) :-
        journey(LastState, NewState, Sts, Sts2),
        \+ visited(NewState, VisitedSeq),
        extend([NewState, LastState | VisitedSeq], Seq, Sts2).
```

Appendix B

Puzzle Solver in pyDatalog

```
pyDatalog.create_terms('safe, Bank, f, g, c, w')
safe(Bank) <= f._in(Bank)
safe(Bank) <= ~g._in(Bank)
safe(Bank) <= g._in(Bank) & ~c._in(Bank) & ~w._in(Bank)

pyDatalog.create_terms('goal, S')
goal([[], S]) <= f._in(S) & g._in(S) & w._in(S) & c._in(S)

def count(list):
    fc, gc, cc, wc = 0, 0, 0, 0
    for i in list:
        if i == f:
            fc += 1
        elif i == g:
            gc += 1
        elif i == c:
            cc += 1
        else: wc += 1
    return [fc, gc, cc, wc]
pyDatalog.create_terms('equiv, count, N1, S1, N2, S2')
equiv([N1, S1], [N2, S2]) <= (count(N1) == count(N2)) &\
                             (count(S1) == count(S2))

pyDatalog.create_terms('visited, St1, St2, L')
visited(St1, L) <= St2._in(L) & equiv(St1, St2)

def headAndTail(list):
    if not list:
        return []
    else:
        return [list[0], list[1:]]
pyDatalog.create_terms('headAndTail, remove, Head, Tail, X, Rest, NewRest')
remove(Head, L, Tail) <= ([Head, Tail] == headAndTail(L))
remove(X, L, Rest) <= ([Head, Tail] == headAndTail(L)) &\
                      remove(X, Tail, NewRest) &\
                      (Rest == [Head] + NewRest)

pyDatalog.create_terms('choose, Items, NewBank, TempBank, chooseRest, RestItems')
choose(Items, Bank, NewBank) <= (f == headAndTail(Bank)[0]) &\
                                       remove(f, Bank, TempBank) &\
                                       chooseRest(RestItems, TempBank, NewBank) &\
                                       (Items == [f] + RestItems)

chooseRest([], Bank, Bank) <= safe(Bank)
chooseRest([X], Bank, NewBank) <= remove(X, Bank, NewBank) & safe(NewBank)

pyDatalog.create_terms('journey')
journey([N1, S1], [N2, S2]) <= choose(Items, N1, N2) & (S2 == Items + S1)
journey([N1, S1], [N2, S2]) <= choose(Items, S1, S2) & (N2 == Items + N1)

pyDatalog.create_terms('succeeds, extend, Seq')
succeeds(Seq) <= extend([[[f, g, w, c], []]], Seq)

def reverse(list):
    return list[::-1]

pyDatalog.create_terms('Sts, NewSts, St, reverse')
extend(Sts, Seq) <= (St == headAndTail(Sts)[0]) & goal(St) &\
                    (Seq == reverse(Sts))
extend(Sts, Seq) <= ([St1, Tail] == headAndTail(Sts)) &\
                    journey(St1, St2) &\
                    ~visited(St2, Tail) &\
                    (NewSts == [St2]+Sts) &\
                    extend(NewSts, Seq)
```

Appendix C

Puzzle Solver in `LogicAPI`

```
class safe(Term): pass
Bank = Var('Bank')

safe(Bank) <= \
    ~(
        ~member('f', Bank) &
        member('g', Bank) &
        ~(
            ~member('c', Bank) &
            ~member('w', Bank)
        )
    )

class goal(Term): pass
+goal([[],_])


class count(Func):
    def function(self, l):
        f, g, c, w = 0, 0, 0, 0
        for i in l:
            if i == 'f':
                f += 1
            if i == 'g':
                g += 1
            if i == 'c':
                c += 1
            if i == 'w':
                w += 1
        return [f, g, c, w]

class equiv(Term): pass
N1 = Var('N1')
S1 = Var('S1')
N2 = Var('N2')
S2 = Var('S2')
equiv([N1, S1], [N2, S2]) <= (count(N1) == count(N2)) &\
                             (count(S1) == count(S2))
```

```
class visited(Term): pass
St1 = Var('St1')
St2 = Var('St2')

visited(St1, [St2]+_) <= equiv(St1, St2) & Cut()
visited(St1, [_]+L) <= visited(St1, L)

class remove(Term): pass
Head = Var('Head')
Tail = Var('Tail')
Rest = Var('Rest')
NewRest = Var('NewRest')

+remove(Head, [Head]+Tail, Tail)
remove(X, [Head]+Tail, [Head]+NewRest) <= remove(X, Tail, NewRest)
# print list(query(remove(X, [1,2,3], L)))

class choose(Term): pass
class chooseRest(Term): pass
Items = Var('Items')
NewBank = Var('NewBank')
TempBank = Var('TempBank')

choose(['f']+Items, Bank, NewBank) <= remove('f', Bank, TempBank) &\
                                      chooseRest(Items, TempBank, NewBank)

chooseRest([], Bank, Bank) <= safe(Bank)
chooseRest([X], Bank, NewBank) <= remove(X, Bank, NewBank) & safe(NewBank)

class journey(Term): pass
journey([N1, S1], [N2, S2]) <= choose(Items, N1, N2) & (S2 == Items + S1)
journey([N1, S1], [N2, S2]) <= choose(Items, S1, S2) & (N2 == Items + N1)

class succeeds(Term): pass
class extend(Term): pass
Seq = Var('Seq')
St2 = Var('St2')
succeeds(Seq) <= extend(([[['f', 'g', 'w', 'c'], []]]), Seq)

class reverse3(Func):
    def function(self, l):
        return l[::-1]

extend([St1]+L, Seq) <= goal(St1) & (Seq == reverse3([St1]+L)) & Cut()
extend([St1]+Tail, Seq) <= journey(St1, St2) & ~visited(St2, Tail) &\
                           extend([St2, St1]+Tail, Seq)
```

Appendix D

Python Codes for Visualization

```
results = query(succeeds(Seq))
path = []
prev = None
for result in results:
  solution = result[Seq]
  for st in solution:
      [n, s] = st
      n = set(n)
      s = set(s)
      if prev != None:
          (np, sp) = prev
          boat = (n - np) | (s - sp)
          path += [[n & np, boat, s & sp]]
      path += [[n, set([]), s]]
      prev = (n, s)
print path

import pygame
from sys import exit
f,w,g,c = 'f', 'w', 'g', 'c'
gen = (x for x in path)
images = {}
images[f] = pygame.image.load("images/f.png")
images[w] = pygame.image.load("images/w.png")
images[g] = pygame.image.load("images/g.png")
images[c] = pygame.image.load("images/c.png")
pygame.init()
pygame.display.set_caption("River␣Crossing␣Puzzle")
river = 500
width, height = 800, 400
box = 100
bank = 10
screen = pygame.display.set_mode((width, height))
done = False
x, y = (width - river) / 2 - bank, 350
boatHeight, boatWidth = 10, 140
back = False
finished = False
still = True
```

```
speed = 20
info = gen.next()
# x_transport, x_north, x_south;
def draw():
    [north, trans, south] = info
    tempX = ((width - river) / 2 - box) / 2
    tempY = height
    for i in north:
        tempY -= box
        screen.blit(images[i], (tempX, tempY))
    tempX = (width + river) / 2 + ((width - river) / 2 - box) / 2
    tempY = height
    for i in south:
        tempY -= box
        screen.blit(images[i], (tempX, tempY))
    tempX = x + (boatWidth - box) / 2
    tempY = y
    for i in trans:
        tempY -= box
        screen.blit(images[i], (tempX, tempY))
    for i in south:
        pass

while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
          if finished:
              pygame.quit()
              exit()
          if still:
              try:
                  info = gen.next()
              except StopIteration:
                  finished = True
                  pygame.quit()
                  exit()
              still = False
    screen.fill((220, 220, 220))
    pygame.draw.rect(screen, (0, 128, 255),
                     pygame.Rect((width - river) / 2, 0, river, height))
    pygame.draw.rect(screen, (0, 0, 0),
                     pygame.Rect((width - river) / 2 - bank, 0, bank, height))
    pygame.draw.rect(screen, (0, 0, 0),
                     pygame.Rect((width + river) / 2, 0, bank, height))
    pygame.draw.rect(screen, (128, 100, 80),
                     pygame.Rect(x, y, boatWidth, boatHeight))
    draw()
    if finished or still:
        pygame.display.flip()
        continue
    if back:
        x -= speed
    else:
        x += speed
    if x < (width - river) / 2 - bank:
        back = False
        x = (width - river) / 2 - bank
        still = True
        info = gen.next()
    elif x > (width + river) / 2 + bank - boatWidth:
        back = True
        x = (width + river) / 2 + bank - boatWidth
        still = True
        info = gen.next()
    pygame.display.flip()
pygame.quit()
exit()
```