OO++: Exploring the Multiparadigm Shift

Seth James Nielsen  
Computer Science Department  
Rice University  
Houston, TX 77005  
sethn@cs.rice.edu

Charles D. Knutson  
Computer Science Department  
Brigham Young University  
Provo, UT 84601  
knutson@cs.byu.edu

ABSTRACT
Programming languages and the programming paradigms they embody co-evolve over time. Within industrial and academic circles, for example, object-oriented programming has evolved from and effectively replaced imperative programming. More recently, many object-oriented languages have assimilated features from other programming paradigms, evolving into multiparadigm languages we refer to as “object-oriented plus-plus” or OO++. In this paper we survey the capabilities of six OO++ languages, present OO++ code samples in Python, and propose key characteristics of an OO++ programming paradigm.

1. INTRODUCTION
In the software development process, choosing a programming paradigm, and a programming language to support that paradigm, is a decision that affects all aspects of software creation including design strategies, implementation, and testing (see [3] for a pertinent example). The Sapir-Whorf hypothesis would suggest that the language used to implement a software solution may influence the manner in which a developer analyzes and understands potential solutions.

Programming languages and programming paradigms evolve over time. Evolution within a community is characterized by a widespread language usage and must not be confused with isolated experimentation and theoretical research. Obviously, change within a community begins with an idea investigated by an individual or small group. However, we are interested in demonstrable trends that gain traction within the software development community at large.

An obvious example of this evolution is the shift from purely imperative programming to object-oriented programming. While object-oriented theory has existed since germinating in the 1960’s [2], widespread use of object-oriented languages did not begin until the late 1980’s [10]. At present, object-oriented languages are a de facto standard for most industrial and academic software development and almost all languages support the object-oriented paradigm or have an object-oriented variant¹.

More recently, there has been a noticeable shift from object-oriented programming to multiparadigm programming. Many modern “object-oriented” languages are evolving, or have evolved into, multiparadigm languages that include language features from non-object-oriented paradigms such as functional and logic programming. Such evolution has been facilitated either by native language features or through language extensions. These languages represent a special subset of all multiparadigm languages which share in common the support for object-oriented programming. To differentiate this subset of multiparadigm languages, we have chosen to identify them as object-oriented plus plus or OO++ languages.

In this paper we briefly survey a number of OO++ languages and examine one of these languages (Python) in detail. Next, we identify key aspects of the OO++ programming paradigm. After presenting our conclusions, we also include two appendices at the end of this paper with technical information regarding multiparadigm extensions to Python.

2. A SURVEY OF OO++ LANGUAGES
Historically, the imperative, functional and logic programming paradigms have been the most widely used paradigms next to object-oriented [4] and our survey of OO++ languages is consistent with this². We found that, in these languages, functional programming is the most popular secondary paradigm (a “graft” into an otherwise object-oriented language) followed by logic programming. Other paradigms such as aspect-oriented and visual programming have benefited from active research communities, but are not as widespread, particularly in commercial off-the-shelf programming languages.

Among OO++ languages, we find some that natively support multiparadigm features. Python and Perl, for example, are two popular scripting languages that support the object-oriented paradigm and aspects of functional programming

¹Including object-oriented versions of such standards as Prolog, Fortran, Cobol, Lisp and Pascal
²Since the object-oriented programming paradigm largely subsumes imperative programming (via data and code encapsulation in objects), we omit further discussion of imperative as an independent paradigm.
natively. These languages also support logic programming and additional functional language features through extension modules.

Ruby is another OO++ language that natively supports functional programming features. Ruby differs from Python and Perl in that object-orientation was the primary design goal and not an add-on to imperative programming. Despite the language's drive for "pure object-orientation", it supports anonymous functions and closures, which are definitely functional concepts.

C++ is also an OO++ language. While primarily object-oriented, C++ supports generic programming, a concept based in functional programming. Additionally, a number of libraries have been written for C++ that provide further multiparadigm language support. For example, FC++ [9] and LC++ [8] are two third-party libraries that augment C++ with extensive functional and logic programming capabilities.

Even some strictly object-oriented programming languages have been refitted into an OO++ variant. The Pizza language [6], for example, is a multiparadigm variant of Java. Pizza augments the Java programming language with functional constructs such as first-class functions, parameterized classes, and algebraic types. Pizza byte-code can operate on any Java virtual machine and the Pizza compiler can convert Pizza source code into standard Java.

Finally, Leda was designed from the start to be truly multiparadigm, seamlessly incorporating object-oriented, imperative, logic and functional programming features into a composite language that integrates the benefits of each. Because Leda was designed to be a multiparadigm language, it excels at supporting collaborations between the various paradigmatic features.

### 3. OO++ PYTHON

In this section we explore OO++ language features of Python in greater detail. Python is an OO++ language with dynamic semantics. To illustrate OO++ code in Python, we first provide examples of functional and logic-programming constructs by themselves before presenting some examples of hybrid feature combinations. The technical detail of the non-native features is described in two appendices at the end of this paper.

In the Python code below, we demonstrate the following functional concepts: closures, self-referential anonymous functions, and higher-order functions. This is a small sample of Python's functional capabilities (both native and extended), but it is sufficient to demonstrate the breadth of functional support. Intricacies specific to Python are explained in code comments (text following the '##' character).

```
# A Simple List Iterator. This demonstrates closures
# in Python. The one caveat of a closure in Python is
# that it only complex variables (like lists) maintain
# state across multiple calls to the function.
# For this reason we use the first element of a list
# (the listPosition variable) instead of a simple
# variable to store the current list position

def makeListIterator(someList):
    listPosition = [-1] # The brackets mean list
    def nextFunction():
        listPosition[0] += 1
        return someList[listPosition[0]]
    return nextFunction

list1 = [2,4,6,8]
next = makeListIterator(list1)
next() # Returns 2
next() # Returns 4

# A self-referential lambda function. The "thisFunction"
# construct is not native to Python. It was
# implemented by the authors of this paper.

factorial = lambda x:((x==0) or (x*thisFunction(x-1))
factorial(3) # Returns 6
factorial(4) # Returns 24

# Finally, currying and compose are two examples
# of higher-order functions. While these two
# functions are not native to Python, their
# implementation (shown in Appendix A) is trivial.

plus = lambda x,y: x+y
plus(2,4) # Returns 6
plus2 = curry(plus, 2)
plus2(5) # Returns 7

square = lambda x: x*x
add_and_square = compose(square, plus) # square(plus(args))
add_and_square(3,2) # Returns 25
```

Logic programming in Python, unlike functional programming, is not supported natively. We have implemented a Prolog-like module that supports logic constructs. Following in typical Prolog fashion, we demonstrate the logic module with a family relationships example.

```
parent = Relation()
parent.assert("Seth","Amy","Alex")
parent.assert("James","Aloyes","Seth")
parent.assert("Lloyd","Dorthy","James")

ancestor = Relation()
# The base case is that a parent is an ancestor
ancestor.inference += lambda ancestor, kid: (parent.query(ancestor, other, kid) | parent.query(other, ancestor, kid))

# The recursive case is that a parent of an ancestor is an ancestor.
ancestor.inference += lambda ancestor, kid: (parent.query(ancestor, temp) & ancestor.query(temp, kid))

X,Y = logicVar(), logicVar()
while ancestor.query(X,Y,Z):
    print X, "is ancestor of", Y
```

Notice that the ancestor relationship is recursive. The output of the preceding script is shown below:
We now examine hybrid-programming constructs by presenting two simple examples.

The first example is an object-oriented/functional hybrid that we term method composition. *Method composition* is similar to functional composition except that side effects are expected (which are otherwise prohibited in functional theory). If the objects to which the methods belong change state, the behavior of the composite function changes transparently.

Imagine that we are creating a computer strategy game. The game has a playing board with pieces that challenge one another if they attempt to occupy the same square. The process for moving a playing piece might be to register the move, resolve challenges (if any), update the game database, and update the screen. We could create this operation as a combination of the sub-operations:

```
move = compose(screen.update, db.update, engine.challenge, game.move)
```

# The next two lines are equivalent
```
screen.update(db.update(engine.challenge(game.move(piece, newX, newY))))
move(piece, newX, newY)
```

Notice that the `move` function is based on the state of the objects. If, for example, the user changes the preferences for the engine (e.g., tougher opponents, etc.) the `move` function is altered transparently.

The second example is a combination of the object-oriented and logic programming paradigms that we term *Non-deterministic Object Construction*. Instead of explicitly declaring the object we want constructed, we describe it by constraints. Our example scenario is a strategy game in which a player competes against the computer in a contest. In such a scenario, the player might wish to describe an opponent instead of requesting one. In the example below, the user competes against all opponents that meet his preferences.

```
Define = Relation()
Define.assert(KnightClass, Attack(10), Armor(10), Move(2))
Define.assert(SamuraiClass, Attack(3), Armor(2), Move(1))
Define.assert(BoxerClass, Attack(5), Armor(1), Move(5))
Define.assert(NinjaClass, Attack(5), Armor(1), Move(10))
Prefs = ui.getUserPreferences()
LogicConstructor = Relation()
LogicConstructor.inference += lambda opponentClass: (
    Define.query(opponentClass, At, Ar, M) &
    Logic_operators.GT(Prefs.attack(), At) &
    Logic_operators.GT(Prefs.armor(), Ar) &
    Logic_operators.GT(Prefs.movement(), M))
badGuy = LogicVar()
while LogicConstructor.query(badGuy):
    badGuyClass = badGuy.value()
    badGuyInstance = badGuyClass()
    engine.fight(player, badGuyInstance)
```

4. KEY ASPECTS OF THE OO++ PARADIGM

Before attempting to characterize the OO++ paradigm, we emphasize the difference between a programming paradigm and a programming language. The former is a guiding philosophy in the creation of software while the latter is a collection of language tools for describing the software to the underlying computing platform. While, theoretically, any programming paradigm could be used to guide the creation of software in any language, in practice a developer uses a language that provides language features that reflect the philosophy of the paradigm. Indeed, “a language that supports a paradigm well is often hard to distinguish from the paradigm itself” [1].

A paradigm can be defined as a set of assumptions, concepts, values and practices that constitute a way of viewing reality [7]. We propose a characterization of the OO++ paradigm by describing key ideas in each of these areas. Our proposed programming paradigm is based on previous multiparadigm research.

Assumptions (*Assumed without Proof*):

- In OO++ languages, the object-oriented paradigm is always fully supported but support for secondary paradigms vary.
- A given programming paradigm leads more naturally to a solution than others for certain identifiable classes of problems.
- Multiple paradigms can be combined effectively within an object-oriented framework.

Concepts (*Ideas Derived from Instances*):

- The object-oriented paradigm is suitable for construction of complex internal program structures and abstractions.
- Some secondary paradigm solutions can be hidden within methods and objects.
- Some secondary paradigm solutions can be integrated with object-oriented structures and abstractions.
Values (Principles Considered Desirable):

- Multiparadigm solutions can be created that are more elegant, readable, and/or simpler than the alternative object-oriented solution.
- Secondary paradigms can be used to generate novel concepts and solutions.

Practices (Customary Actions):

- The object-oriented paradigm can be used to analyze and design structures, states, and interactions.
- Multiple paradigms can be used to design data, operations, and interfaces.
- Language features can be combined into multiple paradigm solutions.
- Language features can be combined into hybrid multiparadigm solutions.

We now explore each of these areas in greater detail.

4.1 Assumptions

An assumption is a principle that appears true and is accepted without proof. Similar to a mathematical axiom, assumptions provide a foundation for reasoning or decision-making and are essential to a programming paradigm. In the following paragraphs, we identify three critical foundational assumptions of the OO++ paradigm.

The first, and most critical, assumption is that in OO++ languages, the object-oriented paradigm is always fully supported while support for secondary paradigms may vary. All other characteristics of OO++ are affected by this assumption. It should be noted that this assumption holds true in the languages we surveyed earlier in this paper.

Secondly, we assume that for certain identifiable classes of problems a given programming paradigm may lead more naturally to a solution than other paradigms. Some of this information is available within the body of research investigating programming paradigms and the types of problems they solve.

Our third assumption is that multiple paradigms can be combined effectively within an object-oriented framework. Obviously the combination of multiple paradigms can result in a “paradigm soup” that is difficult to comprehend and utilize. We assert that the well-structured object-oriented paradigm is capable of integrating and incorporating other paradigms in a usable and comprehensible form.

4.2 Concepts

Whereas assumptions are foundational, the concepts are the building blocks. Not surprisingly, they are often directly related to the assumptions and sometimes overlap with them (assumptions can be concepts, for instance). But while assumptions are generally accepted without proof, a concept can be demonstrated. We identify three key conceptual building blocks of the OO++ paradigm.

The first concept is that the object-oriented paradigm is suited for construction of complex internal program structures and abstractions. The object-oriented paradigm is unique in that it provides numerous constructs specifically for defining and describing structure. Inheritance, information hiding, and encapsulation are examples of such features. In OO++ programming, the object-oriented paradigm is explicitly exploited for these structural definition features.

The second concept is that some secondary paradigm solutions can be embedded within methods and objects and provide solutions to sub-problems. These problems, though small from an architectural perspective, may be daunting within their own limited contexts. By hiding these solutions within an object-oriented system, the “paradigm soup” challenge we referred to previously can be obviated.

We observe that the object-oriented paradigm subsumes the imperative paradigm. Within object-oriented programming, imperative concepts and structures are incorporated into the powerful architectural constructs of object classes and instances. With this in mind, the idea of allowing additional paradigms such as functional and logic programming to coexist within object-oriented structures is reasonable.

The final concept we identify is that some secondary paradigm solutions can be integrated with object-oriented structures and abstractions. In these cases, multiparadigm solutions cannot be hidden within object-oriented abstractions because they are designed to augment and improve these abstractions. Non-Deterministic Object Construction, which we illustrated in the previous section for instance, is an example of this concept.

4.3 Values

While concepts and assumptions contribute necessary components to building complex thinking, values provide a paradigm with a framework for that building process by identifying desirable attributes at microcosmic and macrocosmic level. We suggest two such values within the OO++ paradigm.

The first value is the creation of multiparadigm solutions that are more elegant, readable, and/or simpler than the object-oriented solution. While this value may seem obvious, we emphasize what is not valued: speed, efficiency, and other performance factors. Also note that when multiparadigm solutions do not improve elegance, readability or simplicity in a software solution, then a purely object-oriented approach may be desirable.

The second value is the use of secondary paradigms to generate novel concepts and solutions. The first value we described focuses on improving specific software solutions. This value, on the other hand, drives improving problem compre-
hension and expanding solution possibilities for software development in general. Obviously, many novel solutions will be discarded because they are unviuable, but even these may lead to better iterations or reincarnations that are effective.

4.4 Practices

Built on assumptions, concepts and values, a paradigm also includes practices that describe customary acts or common actions within the paradigm context. Obviously there is no way to describe all possible practices within a paradigm, but certain key practices help to define the paradigm. We identify four such OO++ practices.

The first practice is to use the object-oriented paradigm to analyze and design structures, states, and interactions. These components are important at all levels of design but focus on what the design is rather than how it works. We note that object-oriented design is a relatively mature discipline and fairly well understood. Designing an object-oriented framework (either informally, or formally) provides a stable structure for inserting multiparadigm components.

The second practice is to use multiple paradigms to design data, operations and interfaces. While these three design issues overlap somewhat with the architectural design described in the preceding paragraph, their full exploration happens in later stages of design. When creating an architectural design, knowing that two objects interface is sufficient; how those objects interact is determined later.

Each programming paradigm defines data, operations and interfaces differently. Within the object-oriented paradigm, for example, data generally consists of simple and complex variables, and instances of classes. Operations are generally contained in methods, and the invocation of methods and the structure of classes are the interfaces. For a more complete discussion of the ways in which different paradigms conceptualize data, operations, and interfaces, see [4].

Data, operations, or interfaces from one or more secondary paradigms can replace object-oriented counterparts to improve the software solution. Determining when and how to make replacements requires an understanding of the secondary paradigms and an understanding of how the language features will interact with each other. In many cases, experimentation is necessary to generate effective solutions.

The third practice is a method of replacing object-oriented components resulting in multiple paradigm solutions. These solutions are characterized by each paradigm being used in such a way that it “is preserved within some narrow context.” In other words, component pieces of data, operations or interfaces are still identifiable as belonging to one paradigm or another, even though they interact with data, operations or interfaces from a different paradigm. Intercommunication between paradigm components without “dramatically compromising the paradigmatic purity of either” is generally the easiest way to combine paradigms.

The final practice is the other solution for replacing object-oriented components resulting in multiparadigm solutions. Multiparadigm solutions are not “paradigmatically pure” but combine elements from various paradigms to create a new hybrid synthesis. In this type of solution, the data, operations or interfaces may not be easily identifiable as belonging to a specific programming paradigm. In summary, a multiple paradigm solution is a combination of independent subcomponents each derived from individual paradigms, whereas a multiparadigm solution is an integral derivation from a hybrid paradigm and cannot be subdivided into separate paradigmatic components.

5. CONCLUSIONS

Multiparadigm languages are gaining traction in the software development community. We have observed that most modern “object-oriented” languages support at least one other paradigm either natively or through extension modules. We have chosen to call these multiparadigm object-oriented languages OO++. Python is a good example of an OO++ language, supporting multiparadigm features both natively and through extension modules.

In this paper we have proposed an OO++ programming paradigm in terms of assumptions, concepts, values and practices. Our OO++ paradigm is built around an object-oriented core for architectural design and multiparadigm solutions to solve specific solutions within a narrow context. Multiparadigm solutions generally replace an object-oriented data, operations, or interface component to improve readability or simplicity. The results of these replacements are either multiple paradigm (interacting, but paradigmatically pure) solutions or multiparadigm (integral, hybrid) solutions.

6. REFERENCES


In this reference, the author uses the terms data, operative units, and access points.
APPENDIX

A. PYTHON FUNCTIONAL BACKGROUND

Python is an object-oriented language with dynamic semantics. Python also supports the functional concepts of first-class functions, and polymorphic typing. Additionally, map, filter, and other list processing functions are natively provided.

Functions in Python can be defined to accept two classes of arguments. The first are known as arguments and the second are known as keyword arguments. When defining a function, a specific number of arguments can be enumerated and named or a list variable preceded by a single asterisk can be defined that accepts multiple arguments. Keyword arguments are always passed to a special Python dictionary object identified during definition by a preceding double-asterisk. Argument type is never explicitly declared and functions must provide their own type-checking if required. Functions and classes are legitimate data types. Functions optionally return data.

The code sample below illustrates these characteristics

```python
def simpleFunction():
    print "This function accepts nothing"
    print "and returns nothing."

def simpleFunctionWithArgs(arg1, arg2):
    print "Function called with ", arg1, arg2
    return 1

def moreComplicatedFunction(arg1, arg2, **kargs):
    # The 'kargs' argument is the keywords argument
    # & dictionary
    # & dictionary is a unique-key/values pair Python object
    print "Function called with ", arg1, arg2
    print "Keywords/Values:", for key in kargs.keys():
        print key, "/", kargs[key]

# moreComplicatedFunction(1,2,X=3,Y=4) outputs:
# Function called with 1 2
# Keywords/Values:
# Y / 4
# X / 3

def variableArgsFunction(*args):
    # The 'args' argument is a python list object
    print "Function called with", args

# variableArgsFunction(1,"hi",3.4) outputs:
# Function called with (1,"hi",3.4)

def complexFunction(*args, **kargs):
    print "Function called with", args
    for key in kargs.keys():
        print key, "/", kargs[key]

# complexFunction combines variable arguments
# and keyword arguments
```

B. HIGHER-ORDER FUNCTION EXTENSIONS

Because Python allows functions to be passed and returned as data, creating higher-order functions is trivial. Below we define compose and curry in Python.

```python
def compose(func1, func2):
    def composition(*args, **kargs):
        return func1(func2(*args, **kargs))
    return composition

def curry(originalFunction, *curriedArgs):
    def curriedFunction(*args, **kargs):
        return originalFunction(*(curriedArgs+args),**kargs)
    return curriedFunction
```

C. SELF-REFERENTIAL LAMBDA CALLS

In Python, the dynamic nature of the language allows subroutines to introspect the interpreter to discover information about the call stack. We exploited this introspective ability to load the code of the calling subroutine and recreate the unnamed function.

```python
import new, sys

def thisFunction(*args, **kargs):
    previousCodeFrame = sys._getframe(1)
    callingFunction_CodeObject = previousCodeFrame.f_code
    callingFunction = new.function(
        callingFunction_CodeObject,
        globals())
    return callingFunction(*args, **kargs)
```

6Of course, this is more of a pseudo-curry function. This function "freezes" a given argument. The true functional definition of curry is more complex.
There are two major deficiencies to our extension. The first is that this extension is not guaranteed to be compatible with every version of Python. This is because it draws heavily on the current implementation of internal Python representations of code and functions. The other problem is that every call to thisFunction recreates the original function, thus wasting memory and degrading performance. In the future, we hope to create a more generic, efficient self-referential function.

D. PYTHON LOGIC BACKGROUND

Logic programming is not supported at all in Python, requiring the implementation of a logic module from scratch. In creating such a module, we implemented a Prolog-like pseudo-syntax. We use the term “pseudo-syntax” to clarify that we will not modify the Python interpreter or language in any way. Instead, we will use existing Python syntax in a unique way to give the appearance of a new Prolog-like syntax.

In logic programming there are two distinct operations: defining data and querying. The syntax for defining data is relatively straightforward and maps well to Prolog syntax. Creating querying syntax, on the other hand, is more difficult because in Python, logic constructs must integrate with imperative/object-oriented features for which there are no counterparts in Prolog.

E. DEFINING LOGIC PROGRAMMING FACTS

The most basic data definition in Prolog’s version of logic programming is a fact. In Prolog, facts are defined within relationships explicitly, as in this example.

\[
\text{Father}(\text{Seth}, \text{Alex}) \\
\text{Father}(\text{James}, \text{Seth})
\]

This Prolog code declares that Seth is the father of Alex and that James is the father of Seth.

Our Python based pseudo-syntax is similar.

\[
\text{Father} = \text{Relation}() \\
\text{Father}.\text{assert}(\text{“Seth”, “Alex”}) \\
\text{Father}.\text{assert}(\text{“James”, “Seth”})
\]

Observe our use of the method “assert”. In Prolog, there is a definition phase and a query phase. Data can be defined in the query phase using an “assert” command. In Python there are no forced distinctions for a definition phase and a query phase, hence we always use “assert” to declare facts.

Within our Relation object we store all the elements of each assertion in a unique Python list and then index the elements using Python dictionaries. These dictionaries, which use a hash table to map unique elements to values, serve to tie together equivalent elements in different lists. In other words, the “Seth” string in our example would exist in two separate lists at index zero and index one respectively. An internal dictionary stores the “Seth” string as well as its two locations within the relationship.

F. DEFINING LOGIC PROGRAMMING INFERENCES

Logic programming inferences are rules that define implicit connections between relationships and facts. In Prolog, data is defined in this manner:

\[
\text{Grandfather(\text{grandfather}, \text{grandson}) :-} \\
\text{Father(\text{grandfather}, \text{son}),} \\
\text{Father(\text{son, grandson})}
\]

This inference for the Grandfather relationship implicitly ties the father of a father to his grandson. The comma in the code translates to an “and” in English.

Unfortunately, our pseudo-syntax for logic programming in Python is a bit awkward because of the constraints in declaring functions on the fly. We use lambdas to dynamically create the logical inference.

\[
\text{Grandfather} = \text{Relation}() \\
\text{Grandfather}.\text{inference} = \text{lambda grandfather, grandson:} \\
\text{Father.query(grandfather, son) \\&} \\
\text{Father.query(son, grandfather)}
\]

It is significant that the lambda function defined above will not operate correctly in Python if it is not attached to the inference of a Relation class. Python requires that all variables be declared before use, which is not the case here. The Relation class is written to intercept assignments to the inference member and dynamically instantiate unknown variables as special LogicVariable instances.

Prolog allows multiple inferences to be assigned to any given relation and our Python logic module is comparable. Any number of inferences can be assigned to a Relation instance by using the += operator.

\[
\text{Grandfather.inference} += \text{lambda}
\]

G. LOGIC PROGRAMMING QUERIES

In the preceding section, we introduced query without explaining or defining it. The query method of the Relation class returns an instance to a special iterator class that we have created. Instances of this class can be used in truth tests and returns true in these tests until the iterator is exhausted.

\[
\text{Father} = \text{Relation}() \\
\text{Father}.\text{assert}(\text{“Seth”, “Alex”}) \\
\text{Father}.\text{assert}(\text{“James”, “Seth”}) \\
\text{X,Y = LogicVariable(), LogicVariable()} \\
\text{while Father.query(X,Y):} \\
\quad \text{print X.value(), Y.value()}
\]

The query method of the Father relationship returns an iterator that returns true as long as legitimate values can be assigned to the LogicVariable instances.

However, it is important to observe that every time the query method is called (within the while loop, for example), an iterator is returned. To maintain state, the query call monitors the variables being passed in and stores them.
Every time the same variables are passed in, the same iterator is returned until the iterator fails (which causes a reset).

These special iterators are also operator-overloaded to handle and and or operations. Inferences are built using these operations as described in the previous section.

When a relation composed of facts and inferences is queried, the relation first generates all possible answers from explicit facts. After all the facts are explored, inferences are used to generate answers in the order in which they were defined.

H. SPECIAL OPERATIONS

Prolog uses a special operation called cut to limit backtracking. In our implementation of logic programming, we support a cut method in the Iterator class. A call with a cut would look like this:

```prolog
Father.query(X,Y).cut()
```

However, we also allow the cut to be parameterized with the number of solutions that can be generated before cutting.

```prolog
Father.query(X,Y).cut(4)  # This will allow 4 solutions.
```

Comparison operators are also part of the Prolog language but because we rely on iterators for our system to work correctly, we require the use of a special set of comparison operators that we created for our Python logic programming module. The GT (greater than), LT (less than), NEQ (not equal), and EQ (equal) operators return iterators that can be used in inference construction.

The EQ operator is especially important because it can be used for logical assignment as well as for comparison. If bound logic variables are passed as parameters, EQ behave as a comparison operator. On the other hand, if an unbound logic variable is passed as one of the arguments, it will be bound to the value of the other parameter.

We also provide a special wrapper for any functions or callable objects. These wrapped functions return an iterator that can be used in inference construction.

```prolog
Relation.doFunction(<function name>, (<parameter list>))
```

This special wrapper allows us to insert print functions and other imperative/object-oriented constructs inside inferences of relationships.