Semantics of Compound Comparison Statement and Generator Function in Lambda Calculus

Sumon Biswas

Iowa State University, Ames, IA 50011, USA
sumon@iastate.edu

Abstract. Python is a widely used programming language in education, science and industry. Many distinctive language constructs made Python easy-to-learn and expressive. However, semantics of some advanced features can create corners for the language such as weak scope resolution. The language weaknesses lead to different behaviors from IDEs and create confusions among developers. In this project, our goal is to study two unique language constructs of Python: compound comparison statement and generator. Compound comparison statement is similar to regular `if` statement but it contains chaining comparison as conditions. Chaining comparison (e.g., `x<10<x*10<100`) is a syntactic sugar which combines two comparison operations into one. Generator is another powerful control-flow construct with one or more `yield` statements which is used for creating user-defined iterators. In this study, we have extended Lambda calculus to implement a core language that includes the above two Python language features. We have also described the details of the syntax, operational semantics and type system of the features. In our implementation, we have used Coq proof assistant so that we can further check the correctness of the construct properties.

1 Introduction

In recent years, Python has become one of the three most used programming languages in Github [1]. The great number of libraries, expressive features and easy-to-learn syntax makes it so popular among developers and scientists. Several third-party tools [2] and interactive development environments (IDE) are often used to write Python code since they provide features like finding bugs, refactoring, variable renaming, code completion etc. These features are directly related to language syntax, semantics and runtime behavior. If the language contains ill-defined constructs, the IDEs can offer wrong suggestions and code can result in error or performance issues. This issue is even more severe in case of a rapid changing language like Python which has many advanced syntactic constructs. Moreover, Python is a runtime language which does not define its types in static time. As a result, it is difficult to find type mismatch and scope resolution bugs before running the program. Therefore, there is further need for studying the advanced constructs of Python.

Python has many unique language features which has made it more expressive. In this study, we have implemented two such unique features of Python language: compound comparison statement and generator function. Both are syntactic sugar of the language. Syntactic sugar is a language construct which does not give more power to the language but using the construct a particular task can be handled more easily with less typing. Compound comparison statement is similar to `if` statement. As the condition of `if` statement, we can put multiple atomic conditions separated by `and` or `or` operator. Using this statement, we can write the conjunction or disjunction of the comparisons with more expressiveness.

Generator function [3] is similar to a regular function but instead of returning a value, it returns an iterator. Generator function contains the `yield` statement instead of `return` to produce a single value of the iterator. The generator function is used to create a generator object. The generator object produces the next value of the iterator when `next()` is called.

In this study, we have extended the Lambda Calculus to implement a core language with the two Python features. Lambda calculus is a tiny language which encodes key features of any programming language like functions, procedures, type systems etc. We have formalized the calculus to show the details of syntax, operational semantics and type system of the features. This allows us to write proofs and check correctness of the language properties. For implementing the core language, we have used Coq proof assistant which provides a nice way of defining language rules as inductively. We have chosen to write the language rules as small-step semantics so that it is possible to take care of each intermediary stages of language evaluaton.

In section 2, we have described the Lambda calculus, small-step relations and type system of our core language. In section 3, the other related works are summarized where the authors tried to study language features and its correctness. In section 4, we have described the syntax, operational semantics and type systems and subtyping rules in detail. Section 6 concludes the study with possible future directions.

2 Background

In this section, we have described the Lambda calculus, abstract syntax, small-step semantic relations, type system and subtyping rules of our core language.

2.1 Lambda Calculus

The simply typed lambda calculus (STLC) is a tiny core language that has all the key concept to implement the modern language features like function, procedures, type system etc. To build new language features or study and prove existing languages we extend Lambda calculus. The STLC is built on the collection of the base types such as: boolean, natural numbers, strings. We can also implement advanced types like pairs, list, maps using the STLC. In addition, STLC gives us a way to define the control structure and functional abstraction using three key features:

– variables
– function abstraction
– application

STLC is a functional language where functions are first order values. All functions are anonymous and they can take other functions as parameters.

2.2 Abstract Syntax

To implement a language, we first define the syntax of the features. The types and the constructors are defined using the inductive definitions. Inductive definitions allow us to use the definition itself in side of the body.

2.3 Small Step Semantic Relations

After defining the syntax, we formalize the runtime behavior of the programs: how a term is evaluated to another term, what are the preconditions of a evaluation and what are the steps. We can formalize the semantics using fixpoint relation or using inductive definition. In this study, we have used the inductive relations. There are two ways of defining the dynamic semantics of the language inductively:

- big step relation
- small step relation

In big step relation, we directly define the result of the evaluation from the given term. On the other hand, in small step relation we define each stage of the evaluation inductively. In this study, we have used the small step inductive relation.

2.4 Type System

Type sytem gives a way to classify the values of the expression into different categories and restrict the rules of evaluation into certain classes. By defining type system, we can analyze programs easily and its easy to fis mistakes while writing a program. Sometimes, types can be inferred by the interpreter in runtime and sometime the programmer needs to write the types of each variable explicitly. Types system also allows us to implement two important properties of programs:

- progress
- preservation.

2.5 Subtyping

In a program there might be some types which can be also considered as another type with larger scope. In those cases the type with smaller scope is a subtype of the type with larger scope. Formally, S is a subtype of T, written as $S <: T$, if a value of type S can be safely used where the program expects a value of type T. The concept of subtyping comes handy when we have multiple parent and child types. Generally, we define a super type which is a parent type of all other type. The other subtyping relations are defined as necessary with the values of each expression.

3 Related Work

Most of the works that analyze Python programs do not study the semantics and runtime of language constructs. The study in [4] and [5] focuses on the similar language study of JavaScript. In [6], the authors implemented a small version of the whole Python language to study the subtle dynamic behavior of the language. However, they did not implement all the features of the language like the object oriented features, scope resolution etc. In [7], the authors build an executable semantics for Python 2.5 and test it against manually written checks. However, this work does not define a core language. Rather, it directly builds the evaluation of terms. Therefore, it is not suitable for foundational study and proofs of the language. In [8], the authors build a mini Python interpreter with states and static value analysis. They provide formal semantics and syntax for most of the basic statements and control structures of Python. However, the implementation is not complete for all the syntax of Python and the authors provide analysis for a few of the constructs.

4 Approach

In this section, we will describe the abstract syntax, operational semantics and type system of our core language. In our implementation, we are considering two unique constructs from Python language: compound comparison statement and generator function. The language implementation are written using Coq proof assistant in `corelang.v` file. For this implementation, we have used the pure simply types Lambda calculus (STLC) definition from [9] and then extended the STLC to implement the two features.

4.1 Compound Comparison Statement

In Python, we can write the conjunction of the comparison with less typing. For example, consider the compariosn `x<10 and x*10>10 and x*10<100` can be written using the chaining comparison: `x<10<x*10<100`. In our implementation, we have defined a comparison operator that can take three terms which in turn is separated into two comparisons and finally perform the `AND` operation on those two comparisons.

For example, consider the following Python syntax:

```
if 10 >= 5 >= 0 :
  do_something()
else
  do_something_else()
```

Compound comparison statement in Python

The above code works same as the following one:

```
if 10 >= 5 and 5 >= 0 :
  do_something()
else
  do_something_else()
```

Regular conditional statement in Python

In this section, we have described the details of abstract syntax, operational semantics, typing system and subtyping rules for the two features.

Abstract Syntax for Compound Comparison In our language, we have considered the greater-equal operation ($t_1 \geq t_2 \geq t_3$) as our chaining comparison. The operator is named `CBGe` which utilized the Boolean operators: greater-equal (`BGe`), equal (`BEq`), and (`BAnd`), not (`BNot`). We also need the regular `test` conditional implementation for chaining comparison. Here, two different conditionals are needed for comparing the Booleans and Natural numbers. The abstract syntax of the core language is shown below in BNF (Backus-Naur Form) notation. The syntax are as follows:

```
t ::=                          Terms
  | ...                            previous terms
  | tru                            constant true
  | fls                            constant false
  | test0 t then t else t          Nat conditional
  | test t then t else t           Boolean conditional
  | BBool t                        Boolean constant
  | BNot t                         negation
  | BAnd t t                       And operation
  | BEq t t                        Boolean equal
  | BGe t t                        Boolean equal
```

Syntax for constants and boolean operations

```
t ::=                          Terms
  | ...                            previous terms
  | CBGe t t t                     chained greater-equal comp.
  | Ctest t t t
    then t else t                  chained test operation
```

Syntax for conditional operations

Semantics for Compound Comparison The chaining operator `Ctest` takes five terms. The last two terms are `then` and `else` block of the statement. The `CBGe` operator performs a comparison among the first three terms. Here, we have considered `and` operation between first and second terms as well as `and` operation between second and third term. The operational semantics for regular boolean test operation:

$$\frac{}{test\ tru\ t_1\ t_2 \rightarrow t_1}\ \text{ST\_TESTTRU}$$

$$\frac{}{test\ fls\ t_1\ t_2 \rightarrow t_2}\ \text{ST\_TESTFLS}$$

$$\frac{t_1 \rightarrow t_1'}{test\ t_1\ t_2\ t_3 \rightarrow test\ t_1'\ t_2\ t_3}\ \text{ST\_TEST}$$

The operational semantics for boolean test operation on natural zero:

$$\frac{}{\mathit{test}\ (\mathit{const}0)\ t_1\ t_2 \to t_1}\ \text{ST\_TESTZERO}$$

$$\frac{}{\mathit{test}\ (\mathit{const}S(n))\ t_1\ t_2 \to t_2}\ \text{ST\_TESTNONZERO}$$

$$\frac{t_1 \to t_1'}{\mathit{test}\ t_1\ t_2\ t_3 \to \mathit{test}\ t_1'\ t_2\ t_3}\ \text{ST\_TEST1}$$

The operational semantics for other boolean operations are as follows:

$$\frac{t_1 \to t_1'}{\mathit{BEq}\ t_1\ t_2 \to \mathit{BEq}\ t_1'\ t_2}\ \text{ST\_BEQ}$$

$$\frac{}{\mathit{BEq}\ (\mathit{const}\ n_1)\ (\mathit{const}\ n_2) \to \mathit{BBool}(\mathit{beq\_nat}\ n_1\ n_2)}\ \text{ST\_BEQNAT}$$

$$\frac{}{\mathit{BEq}\ (\mathit{var}\ s_1)\ (\mathit{var}\ s_2) \to \mathit{BBool}(\mathit{eqb\_string}\ s_1\ s_2)}\ \text{ST\_BEQSTRING}$$

$$\frac{\mathit{value}\ v_1 \qquad t_2 \to t_2'}{\mathit{BEq}\ v_1\ t_2 \to \mathit{BEq}\ v_1\ t_2'}\ \text{ST\_BEQVAL}$$

$$\frac{t_1 \to t_1'}{\mathit{BGe}\ t_1\ t_2 \to \mathit{BGe}\ t_1'\ t_2}\ \text{ST\_BGE}$$

$$\frac{\mathit{value}\ v_1 \qquad t_2 \to t_2'}{\mathit{BGe}\ v_1\ t_2 \to \mathit{BGe}\ v_1\ t_2'}\ \text{ST\_BGEVAL}$$

$$\frac{}{\mathit{BGe}\ (\mathit{const}\ n_1)(\mathit{const}\ n_2) \to \mathit{BBool}(\mathit{ble\_nat}\ n_2\ n_1)}\ \text{ST\_BGECONST}$$

$$\frac{t_1 \to t_1'}{\mathit{BNot}\ t_1 \to \mathit{BNot}\ t_1'}\ \text{ST\_BNOT}$$

$$\frac{}{\mathit{BNot}\ (\mathit{BBool}\ b_1) \to \mathit{BBool}\ (\mathit{negb}\ b_1)}\ \text{ST\_BNOTVAL}$$

$$\frac{t_1 \to t_1'}{\mathit{BAnd}\ t_1\ t_2 \to \mathit{BAnd}\ t_1'\ t_2}\ \text{ST\_BAND}$$

$$\frac{\mathit{value}\ v_1 \qquad t_2 \to t_2'}{\mathit{BAnd}\ \mathit{BBool}\ b_1\ \mathit{BBool}\ b_2 \to \mathit{BBool}(\mathit{andb}\ b_1\ b_2)}\ \text{ST\_BANDBOOL}$$

$$\frac{\mathit{value}\ v_1 \qquad t_2 \to t_2'}{\mathit{BAnd}\ v_1\ t_2 \to \mathit{BAnd}\ v_1\ t_2'}\ \text{ST\_BANDVAL}$$

The operational semantics for chained comparison operations are as follows:

$$\frac{t_1 \to t_1'}{CBGe\ t_1\ t_2\ t_3 \to BGe\ t_1'\ t_2\ t_3}\ \text{ST\_CBGE}$$

$$\frac{value\ v_1 \qquad t_2 \to t_2'}{CBGe\ v_1\ t_2\ t_3 \to BGe\ v_1\ t_2'\ t_3}\ \text{ST\_CBGEVAL1}$$

$$\frac{value\ v_1 \qquad value\ v_2 \qquad t3 \to t_3'}{CBGe\ v_1\ v_2\ t_3 \to BGe\ v_1\ v_2\ v_3'}\ \text{ST\_CBGEVAL2}$$

$$\frac{}{\begin{array}{c}BGe\ (const\ n_1)(const\ n_2)(const\ n_3) \to \\ BAnd(BGe(const\ n_1)(const\ n_2))(BGe(const\ n_2)(const\ n_3))\end{array}}\ \text{ST\_CBGECONST}$$

$$\frac{}{Ctest\ t_1\ t_2\ t_3\ t_4\ t_5 \to test(CBGe\ t_1\ t_2\ t_3)\ t_4\ t_5}\ \text{ST\_CTEST}$$

Type System for Compound Comparison The typing rules for the Boolean and natural conditionals are as follows:

$$\frac{Gamma\ \vdash t_1 \in Bool \qquad Gamma\ \vdash t_1 \in T_1 \qquad Gamma\ \vdash t_3 \in T_1}{Gamma\ \vdash (test\ t_1\ t_2\ t_3) \in T_1}\ \text{T\_TEST}$$

$$\frac{Gamma\ \vdash t_1 \in Nat \qquad Gamma\ \vdash t_1 \in T_1 \qquad Gamma\ \vdash t_3 \in T_1}{Gamma\ \vdash (test0\ t_1\ t_2\ t_3) \in T_1}\ \text{T\_TEST0}$$

The typing rules for other Boolean operations are as follows:

$$\frac{Gamma\ \vdash b_1 \in Bool}{Gamma\ \vdash (BBool\ b_1) \in Bool}\ \text{T\_BBOOL}$$

$$\frac{Gamma\ \vdash t_1 \in T \qquad Gamma\ \vdash t_2 \in T}{Gamma\ \vdash (BEq\ t_1\ t_2) \in Bool}\ \text{T\_BEQ}$$

$$\frac{Gamma\ \vdash t_1 \in T \qquad Gamma\ \vdash t_2 \in T}{Gamma\ \vdash (BGe\ t_1\ t_2) \in Bool}\ \text{T\_BGE}$$

$$\frac{Gamma\ \vdash t_1 \in Bool}{Gamma\ \vdash (BNot\ t_1) \in Bool}\ \text{T\_BNOT}$$

$$\frac{Gamma\ \vdash t_1 \in Bool \qquad Gamma\ \vdash t_2 \in Bool}{Gamma\ \vdash (BAnd\ t_1\ t_2) \in Bool}\ \text{T\_BAND}$$

The compound comparison statement makes use of above rules to define its own type. It comprises of two operator CBGe and CTest. Below, we describe the typing rules for those two constructors.

$$\frac{Gamma\ \vdash t_1 \in Bool \qquad Gamma\ \vdash t_2 \in Bool \qquad Gamma\ \vdash t_3 \in Bool}{Gamma\ \vdash (CBGe\ t_1\ t_2\ t_3) \in Bool}\ \text{T\_CBGE}$$

$$\frac{\begin{array}{c}Gamma\ \vdash t_1 \in Bool \qquad Gamma\ \vdash t_2 \in Bool \qquad Gamma\ \vdash t_3 \in Bool \\ Gamma\ \vdash t_4 \in T_1 \qquad Gamma\ \vdash t_5 \in T_1\end{array}}{Gamma\ \vdash (CTest\ t_1\ t_2\ t_3\ t_4\ t_5) \in T_1}\ \text{T\_CTEST}$$

4.2 Generator Function

Generator function [3] is similar to a regular function but instead of returning a value, it returns an iterator (stream of values). Generator function contains the yield statement instead of return to produce a single value of the iterator. The generator function is used to create a generator object. The generator object produces the next value of the iterator when next() is called.

```
def countdown(num):
  while num > 0:
    yield num
    num -= 1
gen_ob = countdown(5)
```

Generator function in Python

In the above example, a generator function countdown has been defined. By calling the function we get the generator object gen_ob. We can get each value produced by the generator by calling next(gen_ob).

In our implementation, we have created a fix relation with abstraction and application to build generator function which can iterate and produces a list of Nat using an arbitrary function. The formal fixpoint definition of the generator function is given below. It takes a boolean condition and a textttNat value and produces a list of texttNat values. The loop inside the function continues until the boolean condition is true. For producing value any arbitrary function can be used. In this case, we have used the predecessor (pred) function so that the list goes through 0. Note that, the texttNat argument of the function works as a seed. From the produced list, we can extract each of the Nat values by calling the next function.

```
Fixpoint gen (b: bool) (n: nat) : (List nat) :=
  match b with
    | true => n :: test0 (gen (false) (const 0))
            (gen (true) (pred n))
    | false => nil nat
  end.
```

Fixpoint for the generator function

We have implemented the recursion inside the generator using basic STLC abstraction and `Fix` relation.

Syntax for Generator Function First, we define our simply typed Lambda calculus, since we will use it to define our generator relation.

```
t ::=                          Terms
  | ...                          previous terms
  | x                            variable

  | λx:T.t                       abstraction
  | t t                          application
```

Syntax for simply typed Lambda calculus

Next, we define the syntax of the iterator that we will use to hold the values generated by our generator function.

```
t ::=                          Terms
  | ...
  | iterNil t                    nil iterator
  | iterConst t t                concatenate iterator
  | iterHead                     returns the next value
```

Syntax for iterator

```
t ::=                          Terms
  | ...
  | gen t t                      generator function
  | genob t                      generator object
  | next t                       returns the next value
```

Syntax for generator function

Operational Semantics for Generator Function The small step dynamic semantics of the iterators are as follows:

$$\frac{t_1- > t_1'}{iterCons\ t_1\ t_2 \to iterCons\ t_1'\ t_2}\ \text{ST\_ITERCONS1}$$

$$\frac{t_2- > t_2'}{iterCons\ v_1\ t_2 \to iterCons\ v_1\ t_2'}\ \text{ST\_ITERCONS1}$$

$$\frac{t_1- > t_1'}{iterHead\ t_1\ t_2 \to iterCons\ t_1'\ t_2}\ \text{ST\_ITERHEAD}$$

$$\frac{value\ h_1}{iterHead(iterCons\ h_1\ l_1) \to h_1}\ \text{ST\_ITERHEADVAL}$$

The semantics for generator function are as follows:

$$\frac{t_1- > t_1'}{gen\ t_1\ t_2 \to gen\ t_1'\ t_2}\ \text{ST\_GEN1}$$

$$\frac{value\ v_1 \qquad t_2- > t_2'}{gen\ v_1\ t_2 \to gen\ v_1\ t_2'}\ \text{ST\_GEN2}$$

$$\frac{value\ v_1 \qquad value\ v_2}{\begin{array}{c} gen(abs\ t_1\ Bool\ v_1)(abs\ t_2\ Nat\ v_2) \to \\ tfix\ (abs\ f\ (Arrow(Arrow\ Bool\ Nat)(List\ Nat))(abs\ t1\ Bool(abs\ t2\ Nat(test(var\ t1) \\ (tcons(var\ t2)((test0\ (var\ t2) \\ (app\ (var\ f)(app\ (fls)(const\ 0)))(app\ (var\ f)(app\ (tru)(prd\ (var\ t2))))))) \\ (tnil\ Nat))))) \end{array}}\ \text{ST\_GEN}$$

$$\frac{t_1 \to t_1'}{next\ t_1 \to next\ t_1'}\ \text{ST\_GENOB}$$

$$\frac{value\ v_1}{genob\ v_1 \to iterCons\ v_1\ iterNil}\ \text{ST\_GENOBVAL}$$

$$\frac{t_1 \to t_1'}{next\ t_1 \to next\ t_1'}\ \text{ST\_NEXT}$$

$$\frac{}{next\ v_1 \to iterHead(v_1)}\ \text{ST\_NEXTVAL}$$

Type System for Generator Function First, we define the typing rules for the basic STLC which we will use to define our recursive definition for generator.

$$\frac{Gamma \vdash x\ =\ T}{Gamma\ \vdash (var\ x) \in T}\ \text{T\_VAR}$$

$$\frac{(Gamma x \mapsto T_{11}) \vdash t_{12} \in T_{12}}{Gamma\ \vdash (abs\ x\ T_{11}\ t_{12}) \in (Arrow\ T_{11}\ T_{12})}\ \text{T\_ABS}$$

$$\frac{\begin{array}{c}Gamma \vdash t_1 \in (Arrow\ T_1\ T_2)\\ Gamma \vdash t_1 \in T_1\end{array}}{Gamma\ \vdash (generator\ t_1\ t_2) \in T_2}\ \text{T\_APP}$$

The typing rules for the iterator are as follows:

$$\frac{}{Gamma\ \vdash iterNil \in IterNil}\ \text{T\_ITERNIL}$$

$$\frac{Gamma\ \vdash t \in T \qquad Gamma\ \vdash tr \in Tr}{Gamma\ \vdash (iterConsttr) \in (IterConsTTr)}\ \text{T\_ITERCONS}$$

Then, we define the typing rules for the generator, generator object and next call. For, defining the semantics of generator we use fix relation as recursive function.

$$\frac{Gamma \vdash t_1 \in (Arroow\ T_1 T_2)}{Gamma \vdash tfix\ t_1 \in T_1}\ \text{T\_FIX}$$

$$\frac{Gamma \vdash t_1 \in Bool \qquad Gamma \vdash t_2 \in Nat}{Gamma \vdash gen\ t_1\ t_2 \in ListNat}\ \text{T\_GEN}$$

$$\frac{Gamma \vdash t_1 \in ListT_1}{Gamma \vdash genob\ t_1 \in (IterCons\ T_1\ IterNil)}\ \text{T\_GENOB}$$

$$\frac{Gamma \vdash t_1 \in IterCons\ T_1\ T_2}{Gamma \vdash next\ t_1 \in T_1}\ \text{T\_NEXT}$$

4.3 Subtyping

In this section, we describe the subtyping rules we have defined in our core language. We have defined general binary subtyping rules for two or more types and also built a iterator type which can have subtypes. The iterator can hold multiple values having different types. When a new value with new type is added to the iterator, the type of iterator is defined combining all the types of the components.

In our core language, every type is a subtype of the type Top. The relation is as follows:

$$\frac{}{S <: Top}\ \text{S\_TOP}$$

The first rule of subtyping is the rule of subsumption:

$$\frac{Gamma \vdash t \in S \qquad S <: T}{Gamma \vdash t \in T}\ \text{S\_SUB}$$

The second one is the structural rule of subtyping which is independent of any particular types.

$$\frac{S <: U \qquad U <: T}{S <: T}\ \text{S\_TRANS}$$

The third structural rule of subtyping is reflexivity which is also independent of any specific type. It says that every type is a subtype of itself.

$$\frac{}{T <: T}\ \text{S\_REFL}$$

Next, we define the subtyping of the arrow type which defines the rules for passing a subtype as parameter.

$$\frac{S_2 <: T_2}{S_1 \to S_2 <: S_1 \to T_2}\ \text{S\_ARROW\_CO}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}\ \text{S\_ARROW}$$

The subtyping rules for the iterators are as follows:

$$\frac{}{(IterCons\ T_1\ T_2) <: IterNil}\ \text{S\_ITRWIDTH0}$$

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{(IterCons\ S_1\ S_2) <: (IterCons\ T_1\ T_2)}\ \text{S\_ITRWIDTH1}$$

4.4 Auxiliary Functions

We have used three auxiliary functions to implement Boolean Not, Boolean equal and Boolean less-than-equal check. The definitions are as follows:

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

Auxiliary function for the Boolean negation

```
Fixpoint beq_nat n m :=
  match n, m  with
  | O, O => true
  | O, _ => false
  | S n', O => false
  | S n', S m'  => beq_nat n' m'
  end.
```

Auxiliary function for the Boolean equal check

```
Fixpoint ble_nat n m  :=
  match n, m  with
  | O, _ => true
  | S n', O => false
  | S n', S m'  => ble_nat n' m'
  end.
```

Auxiliary function for the Boolean less-than-equal check

4.5 Others

In this section, we present the syntax and semantics taken from [9] which were necessary for our core language implementation.

Abstract Syntax Below we present the types for our core language:

```
T ::=           Type
  | Bool          boolean value
  | Nat           natural numbers
  | List          list of other types
  | Arrow         used for defining functions
```

Syntax for primitive types

The syntax and values for defining the natural numbers are as follows:

```
t ::=                    Terms
  | ...
  | t.const                 number
  | t.scc                   successive number
  | t.prd                   previous number

v ::=                     Values
  | ...
  | v                         natural value
```

Syntax and values for natural numbers

The syntax and values for defining the list data structure are as follows:

```
t ::=                     Terms
  | ...
  | nil T
  | cons t t
  | x::x => t

v ::=                     Values
  | ...
  | nil T                  nil value
  | cons v v               cons value
```

Syntax and values for list data structure

Operational Semantics The operational semantics for the basic STLC and numbers are as follows:

$$\frac{value\ v_2}{(\backslash x : T.\ t_{12}) v_2 \to [x := v_2] t_{12}}\ \text{ST\_AppAbs}$$

$$\frac{t_1 \to t_1'}{app\ t_1\ t_2 \to app\ t_1'\ t_2}\ \text{ST\_App1}$$

$$\frac{\begin{array}{c} value\ v_1 \\ t_2 \to t_2' \end{array}}{app\ v_1\ t_2 \to app\ v_1\ t_2'}\ \text{ST\_App2}$$

$$\frac{t_1 \to t_1'}{scc\ t_1 \to scc\ t_1'}\ \text{ST\_Scc}$$

$$\frac{Const\ n_1}{scc\ n_1 \to S\ n_1}\ \text{ST\_SccNat}$$

$$\frac{t_1 \to t_1'}{prd\ t_1 \to prd\ t_1'}\ \text{ST\_Pred}$$

$$\frac{Const\ n_1}{prd\ n_1 \to pred\ n_1}\ \text{ST\_PredNat}$$

Type System The type system for natural numbers and list data structure are as follows:

$$\frac{}{Gamma\ \vdash (const\ n_1) \in Nat}\ \text{T\_NAT}$$

$$\frac{Gamma\ \vdash t_1 \in Nat}{Gamma\ \vdash (scc\ t_1) \in Nat}\ \text{T\_SUCC}$$

$$\frac{Gamma\ \vdash t_1 \in Nat}{Gamma\ \vdash (prd\ t_1) \in Nat}\ \text{T\_PRED}$$

$$\frac{Gamma\ \vdash t_1 \in Nat \qquad Gamma\ \vdash t_2 \in Nat}{Gamma\ \vdash (mlt\ t_1\ t_2) \in Nat}\ \text{T\_MULT}$$

$$\frac{Gamma\ \vdash t_1 \in Nat \qquad Gamma\ \vdash t_2 \in T_1 \qquad Gamma\ \vdash t_3 \in T_1}{Gamma\ \vdash (test\ t_1\ t_2\ t_3) \in T_1}\ \text{T\_TEST}$$

$$\frac{}{Gamma\ \vdash (tnil\ T) \in (List\ T)}\ \text{T\_NIL}$$

$$\frac{Gamma\ \vdash t_1 \in T_1 \qquad Gamma\ \vdash t_2 \in T_2}{Gamma\ \vdash (tcons\ t_1\ t_2) \in (List\ T_1)}\ \text{T\_CONS}$$

5 Conclusion and Future Work

In this study, we have implemented a core language which includes two important feature of Python programming language. Studying advanced semantics of Python gives us the opportunity to look into the corners of the features. The programming IDEs and language developers can be benefitted from this study. However, we did not prove the properties of those features in this project. By using this implementation, we can write the properties in terms of theory and prove that using inference logic and tactics. Furthermore, we did not look into the other interesting features of Python since Python is a very large language. We can also extend this study in the future by considering different scopes and object oriented features of the language. For example, what will happen while declaring generator function globally and calling the `next()` on the generator object in two different scopes(e.g., classes) consequently. The behavior of the generator function might be different. From our study, we understand that the features can be formalized in many different ways by writing different syntax and semantics. Although, different implementation can behave in the same way in programs, we might leave some caveats. For example, in operational semantics which term of the function is evaluation first can result into different return values of a function.

References

1. Ben Frederickson: Ranking Programming Languages by GitHub Users. (accessed May 8, 2019) `https://www.benfrederickson.com/ranking-programming-languages-by-github-users/`.
2. Ian Lee: Python style guide checker. (accessed May 8, 2019) `https://pypi.org/project/pep8/`.
3. Python Software Foundation: Python 3 Documentation. (accessed April 1, 2019) `https://docs.python.org/3/howto/functional.html#generators`.
4. Politz, J.G., Carroll, M.J., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. In: ACM SIGPLAN Notices. Volume 48., ACM (2012) 1–16
5. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: European conference on Object-oriented programming, Springer (2010) 126–150
6. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: ACM SIGPLAN Notices. Volume 48., ACM (2013) 217–232
7. Smeding, G.J.: An executable operational semantics for python. Universiteit Utrecht (2009)
8. Fromherz, A., Ouadjaout, A., Miné, A.: Static value analysis of python programs by abstract interpretation. In: NASA Formal Methods Symposium, Springer (2018) 185–202
9. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software foundations. Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html (2010)