Project Report: Generator in Lambda Calculus

Yijia Huang[1]

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

Abstract. One of the beauties for programming languages is to optimize massively redundant program instructions in order to lower the implementation effort by human beings. *while* as one of the most significant features in programming languages is used for iterating over a *list* with the repeated instructions. However, memory usage could not be affordable if the stream data contains a large number of values. In that case, we proposed a generator that yields items instead of returning a *list* for memory efficiency. It is very similar to the implementation that built a list in memory but has the memory usage characteristic of the iterator implementation. Nonetheless, if the iteration needs to be performed multiple times, then generating a list in memory might be worth it since making an integer is a very expensive process. In other words, a generator will provide performance benefits only if we do not intend to use that set of generated values more than once and the memory usage of the *list* is reasonable.

1 Introduction

For this course project, I extended the lambda calculus language with the generator feature for memory efficiency in iteration operations.

1.1 Motivation

Generator functions can generate a function that behaves like an iterator. They allow programmers to make an iterator in a fast, easy and clean way without too much memory cost. To illustrate this, let us consider a simple Python example of building a list and return it[1].

```
# firstn with list
def firstn(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums
```

Fig. 1: Firstn with List

In figure 1, the function firstn return a full list with length n in memory. If n is really big number and each integer keeps 10 megabyte in memory, then we need to cost a lot of memory to extend our RAM.

```python
# firstn with generator pattern
class firstn1(object):
    def __init__(self, n):
        self.n = n
        self.num = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num + 1
        else:
            raise StopIteration()
        return cur
```

Fig. 2: Firstn with List

To save memory space, we can implement the firstn1 as an object with generator pattern, fig 2. Class firstn1 is iterable and it will perform as we expect. However, we need to write a bunch lines of code to implement it and the logic is expressed in a convoluted way.

```python
# firstn generator
def firstn2(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

Fig. 3: Firstn Generator

Python provides generator feature in each functions. In the figure 3, the keyword *yield* indicates that the function firstn2 is a generator that yields items in the iteration instead of returning a list. Compare the actually object sizes in these approaches with the same input 10k. The figure 4 shows that the generator has a huge advantage not only in memory efficiency but also in clear and natural logic.

```
In [5]:
1 l, o, g = firstn(10000), firstn1(10000), firstn2(10000)
2 print(sys.getsizeof(l))
3 print(sys.getsizeof(o))
4 print(sys.getsizeof(g))

87624
56
120
```

Fig. 4: Memory Sizes

1.2 Background

In this section, I will show necessary background about this generator functions. For ordinary functions, they have a single entry point and multiple exit points(return statements). When we call a function, the code runs from the first line of the function until it finds an exit point. After that, the function's stack of local variables are cleared and the corresponding memory reclaimed by the OS[2].

However, the generator function have multiple entry and exit points. The function's stack of local variables are allocated on heap memory instead of stack memory. Each *yield* statement will defines an exit point and a re-entry point in the same location. The generator function runs until a *yield* statement is encountered. At that point, the function is paused. And the flow of control is yielded to the caller of the generator function and then back to the re-entry point to resume the function.

```
In [11]:
1 def foo(x,y):
2     print('yielding x')
3     yield x
4     print ('yielding y')
5     yield y
6 g = foo(1, 2)

In [12]:
1 dis.show_code(g)

Name:              foo
Filename:          <ipython-input-11-1e68f5d8ec40>
Argument count:    2
Kw-only arguments: 0
Number of locals:  2
Stack size:        2
Flags:             OPTIMIZED, NEWLOCALS, GENERATOR, NOFREE
Constants:
   0: None
   1: 'yielding x'
   2: 'yielding y'
Names:
   0: print
Variable names:
   0: x
   1: y
```

Fig. 5: Function *foo* with Yield

In the figure 5, variable *g* indicates the generator *foo(1, 2)*. When the CPython compiler finds the *yield* keyword in a function, it sets a GENERATOR flag. Then, the com-

piler returns a generator object which is iterable. If we apply *next* function with the generator, then it returns the values of variable *x, y* in the first two calls respectively as shown in Figure 6.

```
In [13]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
Last Instruction Pointer is at: -1

In [14]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)
             10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5          22 LOAD_FAST                1 (y)
             24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(a) Before Next

```
In [15]: next(g)
yielding x
Out[15]: 1

In [16]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
Last Instruction Pointer is at: 10

In [17]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3       -->  8 LOAD_FAST                0 (x)
             10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5          22 LOAD_FAST                1 (y)
             24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(b) First Next

```
In [18]: next(g)
yielding y
Out[18]: 2

In [19]: print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)
Last Instruction Pointer is at: 24

In [20]: dis.disco(g.gi_code, g.gi_frame.f_lasti)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('yielding x')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)
             10 YIELD_VALUE
             12 POP_TOP

  4          14 LOAD_GLOBAL              0 (print)
             16 LOAD_CONST               2 ('yielding y')
             18 CALL_FUNCTION            1
             20 POP_TOP

  5       -->  22 LOAD_FAST                1 (y)
             24 YIELD_VALUE
             26 POP_TOP
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
```

(c) Second Next

Fig. 6: Iterable Generator Foo

2 Approach

2.1 Exit Point and Re-entry Point

The key idea of the generator function is how to encounter and build exit point and re-entry point in the function. To solve this problem, the first step is to implement sequence and we can write

```
r; r := succ(r)
```

as an abbreviation for

```
(\x:Unit. (r := succ(r)) r
```

Let's define a *yield* term in the sequence as shown below. Suppose this sequence is the body code of one function. If the function is called and the program flow encounters a *yield* term, then the term would be regarded as an *exit* point that can pause the program process to return the content in it and a re-entry point to resume the rest executions of the function.

```
yield r; r := succ(r)
```

To achieve this goal, I designed a naive *tseq* function shown as below that takes two terms *t1* and *t2*. If the *t1* is a *yield* term, then it returns an *exit* pair of terms (t1', (abs "_" Unit t2)), where t1' is the actual content returned at that exit point and (abs "_" Unit t2) is the remainder executions of the function without further reduction.

```
Definition tseq t1 t2 :=
match t1 with
| yield t1' => (t1', (\_:Unit. t2))
| _ => (\_:Unit. t2) t1
end.
```

By applying our *tseq* function, the sequence shown above would return a pair:

```
(r,  (\_:Unit. r := succ(r))
```

However, I implemented the sequence as a *seq* term with the similar structure to *list* in my project instead of the *tseq* function in order to ease sequence concatenation and nested sequence flattening.

2.2 Yield in Iterations

To make sure the *seq* term would work in iterations, more explicitly the second term of the returned *exit* pair should contain the rest iterations of a loop, we need to define the loop recursively with *fixed point*. A *while* loop can be defined as below, where p is a predicate function of input r.

```
while (p(r)) {
  yield r;
  r := succ(r)
}
```

Then we can define a *while* function with the *fixed point*.

```
def while (r) {
  if p(x) do
       yield r;
       r := succ(r);
       while(r)
}
```

In my design, the *while* function returns a *seq* term without further reduction. Suppose a *while* function is already one term in a body sequence of a function as shown below. Since my *while* function returns a *seq* term, we can flatten the nested body sequence into

a one-dimensional sequence. In this way, a nested *yield* term can still trigger a complete *exit* pair $(yield\ r,\ t')$, where the t' is a sequence of $r := succ(r); while(r); t3; t4$ containing the rest program of the function f. More details are provided in the operation semantic section.

```
def f(r) {
  t1;
  t2;
  while(r);
  t3;
  t4
}
```

2.3 Generator Feature

I will show how my generator feature works in my language. First, let's give an example about an ordinary function *F*. In line 2, the *x* is the number argument passing into the function and *x'* is the reference for *x*. In line 3 and 4, we increase and decrease the number by 1 in the reference *x'*. Then in line 5, we apply a *while* term constructed by a predicate function P and a body function B to *x'*. At last, we return the value of under the reference cell *x'* in line 6.

```
1 F = \x: Nat.
2     let x' = (ref x) in
3     x' := succ (deref x');
4     x' := pred (deref x');
5     while P B x';
6     !x'
```

The predicate and body functions in the *while* term are:

```
P = \x':(Ref Nat). (deref x') < 5
B = \x':(Ref Nat). x' := succ !x'
```

Next, let's see an example for a generator function. The generator *G* is similar to the function *F*. However, the generator returns a reference of a unit function with a sequence body instead of a pure sequence term in two main reasons:

1. A generator function should return a reference of a iterable object by given an input.
2. A unit function can prevent the reduction process of the *seq* term in the *ref* term. Because I need the *seq* term only be reduced in the *next* term which is used to iterate each items of a iterable object.

```
1 G = \x: Nat.
2     let x' = (ref x) in
3     ref ( \_: Unit.
4           x' := succ (deref x');
5           yield !x';
6           x' := pred (deref x');
7           yield !x';
8           while P B x';
9           yield !x')
```

The predicate and body functions in the *while* term are:

```
P = \x':(Ref Nat). (deref x') < 5
B = \x':(Ref Nat).
     yield !x';
     x' := succ !x'
```

To make this work, I defined a *gen* term to construct a iterable generator object in type $Itr\ T$ by taking a generator function and a corresponding input. I also implemented a *next* function shown as below in order to iterate each item returned at the *exit* point of a *gen* term. Here, the x, y are the first two items returned from the generator g.

```
let g = gen G x in
let x = next(g) in
let y = next(g) in
(x, y)
```

In the following sections, I provide the design of my implementation including syntax, operation semantics, and typing of this generator feature.

3 Syntax

Abstract syntax is a structure used in the representation of text in computer languages, which are stored in a tree structure as an abstract syntax tree. Here, the abstract syntax supporting for my language feature is provide below.

Syntax:

```
t ::=                   Terms
  | ...                 (other terms same as before)
  | seq t1 t2           sequence
  | ref t               reference
  | deref t             dereference
  | assign t1 t2        assignment
  | loc nat             reference cell location
  | let x=t1 in t2      let-binding
  | fix f               fixed-point operator
  | while p b           while function
  | gen t1 t2           iterable generator
  | yeild t             yeild term
  | iterate t1 t2       iterate operator
  | next t              next operator

T ::=                   Types
  | ...
  | (Itr T)             Iterator type

v ::=                   Values
  | ...
  | loc l               reference value
  | yield t             yield value
```

- **Terms**: The *seq* term is used to execute each terms in a order. The reference term is used to tracking variable value globally, so *ref, deref, assign, loc* are needed. The *let* can bind a name *x* to *t1* while reduce *t2*. The *fix* can be applied to a abstraction in order to return its *fixed point* for recursion. The *while* takes a predicate function and a body function to form a while loop function. The *gen* takes a generator function and a argument used to pass into the generator. The *yield* term is used to trigger the exit point and re-entry point in a *seq* term. The *iterate* operator takes a iterable object's reference and the object as two arguments in oder to return the next item. The *next* term takes a iterable term.
- **Types**: The $next$ operator should only take iterable terms. So, I added a iterator type to identify the iterable generator.
- **Values**: Both *loc l* and *yield t* are value terms without any further reduction.
- **Others**: For the other required terms, types, and values, I am not going to show more details here since they are similar to those terms in chapter *MoreStlc*.

4 Operational Semantics

The operational semantics of a programming language is used to interpret a valid program into sequences of computational steps. And these sequences are the meaning of the program. In this section, I will describe the operational semantics for the generator feature.

4.1 Sequence

The *seq* term has similar structure to the list which is constructed by taking two terms. In my design, a *yield* term would trigger a *exit* pair in the reduction of *seq* in order to iterate each item in a generator. To achieve that, I defined *seq_tm* for checking the *seq* term and *yield_tm* for checking the *yield* term. The *seq_app* function is used for concatenating two *seq* terms into one-dimensional *seq* term since the idea of the *exit* pair only works for unnested *seq*.

- **ST_Seq1**: If t_1 is a *seq* term, then append t_1 to t_2.
- **ST_Seq2**: If t_1 is not a *seq* term and it is reducible, then reduce t_1 to t_1'.
- **ST_Seq3**: If v_1 is a value and a *yield* term, then return *exit* pair.
- **ST_Seq4**: If v_1 is a value and a non-*yield* term, then evaluate t_2.

```
Reduction:
                    seq_tm t1
  ------------------------------------------------  (ST_Seq1)
  seq t1 t2 / st --> seq_app t1 t2 / st

                  not (seq_tm t1)
              t1 / st --> t1' / st'
  ------------------------------------------------  (ST_Seq2)
  seq t1 t2 / st --> seq t1' t2 / st'
```

$$\frac{\texttt{yield\_tm v1}}{\texttt{seq v1 t2 / st --> pair v1 (\_:Unit t2) / st}}\ (\texttt{ST\_Seq3})$$

$$\frac{\texttt{not (yield\_tm v1)}}{\texttt{seq v1 t2 / st --> t2 / st}}\ (\texttt{ST\_Seq4})$$

4.2 Reference

- **ST_RefValue**: If v is a value, then return the length is the state list as its location reference and append v to the list.
- **ST_DerefLoc**: If t is a value ($loc\ L$) and L is less than the length of the state list, then find the value under the reference cell in the list.
- **ST_Assign**: If L is less than the length of the state list, then replace the value under reference cell (loc L) with v_2.

Reduction:

$$\frac{}{\texttt{ref v / st --> loc |st| / st,v1}}\ (\texttt{ST\_RefValue})$$

$$\frac{\texttt{t / st --> t' / st'}}{\texttt{ref t / st --> ref t' / st'}}\ (\texttt{ST\_Ref})$$

$$\frac{\texttt{t / st --> t' / st}}{\texttt{!t / st --> !t' / st'}}\ (\texttt{ST\_Deref})$$

$$\frac{\texttt{L < |st|}}{\texttt{!(loc L) / st --> lookup L st / st}}\ (\texttt{ST\_DerefLoc})$$

$$\frac{\texttt{t1 / st --> t1' / st}}{\texttt{t1 := t2 / st --> t1' := t2 / st}}\ (\texttt{ST\_Assign1})$$

$$\frac{\texttt{t2 / st --> t2' / st}}{\texttt{t1 := t2 / st --> t1 := t2' / st}}\ (\texttt{ST\_Assign2})$$

$$\frac{\texttt{L < |st|}}{\texttt{loc L := v2 / st --> unit / [L:=v2]st}}\ (\texttt{ST\_Assign})$$

4.3 Let

ST_LetValue: If v_1 is a value, then substitute x with v_1 into t_2.

```
Reduction:

                t1 / st --> t1' / st'
-------------------------------------------------  (ST_Let1)
let x=t1 in t2 / st
      --> let x=t1' in t2 / st'

-------------------------------------------------  (ST_LetValue)
let x=v1 in t2  / st
      --> [x:=v1]t2  / st'
```

4.4 Fix

Lambda calculus is a universal model of computation which is well defined in Chapter *Stlc*. Then, we can define a general recursion with this. Let us define the *fixed point* $Y = \lambda f.(\lambda g.f(g\ g))(\lambda g.f(g\ g))$. Then the recursion function f could be defined as:

$$\begin{aligned} Yf &= \lambda f.((\lambda g.f(g\ g))(\lambda g.f(g\ g)))f \\ &= (\lambda g.f(g\ g))(\lambda g.f(g\ g)) \\ &= f((\lambda g.f(g\ g))(\lambda g.f(g\ g))) \\ &= f(\lambda f.((\lambda g.f(g\ g))(\lambda g.f(g\ g)))f) \\ &= f(Yf) = f(f(Yf)) = ... \end{aligned}$$

```
Reduction:

            t1  / st--> t1' / st'
-------------------------------------------------  (ST_Fix1)
      fix t1  / st--> fix t1' / st'


-------------------------------------------------  (ST_FixAbs)
  fix (\xf:T1.t2) / st -->
        [xf:=fix (\xf:T1.t2)] t2  / st
```

4.5 While

Since *while* loop can be implemented in recursion as shown below, then we can implement *while* function either in induction or lambda calculus. Here, P is a predicate function of x and B is a body function of x in the following function, where x is reference type.

```
def while (P,B,x) {
    if P(x) do
        B(x); while(P,B,x)
}
```

Then, the *while* term can be formalized as:

```
while (\x:(Ref T). p) (\x:(Ref T). b) =
fix \f:(Ref T)->Unit.
    \x:(Ref T).
    if (\x:(Ref T). p x)
    do (seq_app (\x:(Ref T). b x) (f x)) else unit
```

In this way, (while P B x) always return a *seq* term or a unit term. So, a complete *exit* pair would be triggered by a *yield* term in the reduction of the *seq* term, even if a sequence is nested with a *while* term that is also nested by a *yield* term. In addition, I force the f abstraction has a $(Ref\ T) \rightarrow Unit$ type, since my *while* function only takes a reference as a input and returns a $Unit$ type.

```
Reduction:

            t1 / st --> t1' / st'
       ---------------------------------------- (ST_While1)
       while t1 t2 / st --> while t1' t2 / st'

            t2 / st --> t2' / st'
       ---------------------------------------- (ST_While2)
       while v1 t2 / st --> while v1 t2' / st'

       ---------------------------------------- (ST_WhileFix)
     while (\x1:(Ref T). p) (\x2:(Ref T). b) / st
     --> tfix (\f:(Ref T)->Unit.
              (\x:(Ref T).
              (if \x1:(Ref T). p x
                  then seq_app (\x2:(Ref T). b x) (f x)
                  else unit))) / st'
```

4.6 Generator

For my generator feature, there are four major terms, *gen, gyield, giterate*, and *gnext*. The *gen* term is used to construct a iterable generator object. The *gyield* term is used for trigger a *exit* pair during the reduction of the *seq* term. The *giterate* term is invoked by the *gnext* in order to update a iterable object's reference and return the next item of the iterable object.

- **ST_Gen3**: If t_1 is a generator function and t_2 is its corresponding input, then apply the function with the input to return a reference of a iterable object.

  ```
  Inductive gen_fun : tm -> Prop :=
  | Tgen_fun : forall x T x' t,
  ```

```
gen_fun (abs x T
           (tlet x' (ref (var x))
                    (ref (abs "_" Unit t)))).
```

Reduction:

$$\frac{\texttt{t1 / st} \longrightarrow \texttt{t1' / st'}}{\texttt{gen t1 t2 / st} \longrightarrow \texttt{gen t1' t2 / st'}}\ (\text{ST\_Gen1})$$

$$\frac{\texttt{t2 / st} \longrightarrow \texttt{t2' / st'}}{\texttt{gen v1 t2 / st} \longrightarrow \texttt{gen v1 t2' / st'}}\ (\text{ST\_Gen2})$$

$$\frac{\texttt{gen\_fun v1}}{\texttt{gen v1 v2 / st} \longrightarrow \texttt{app v1 v2 / st}}\ (\text{ST\_Gen3})$$

– **ST_Gyield**: A *yield* term is a value if only if t is a value.

Reduction:

$$\frac{\texttt{t / st} \longrightarrow \texttt{t' / st'}}{\texttt{gyield t / st} \longrightarrow \texttt{gyield t' / st'}}\ (\text{ST\_Gyield})$$

The first term of the *giterate* term is a iterable object's reference. And the second one is the iterable object in the reference.

– **ST_GiterateExitPair1**: Evaluate the body t of the generator.
– **ST_GiterateExitPair2**: If the body t can be reduced to a *exit* pair, then assign the rest sequences to the generator's reference and return the yield value.
– **ST_GiterateExitPair3**: If the body t can be reduced to a *yield* term, then assign unit to the generator's reference and return the yield value.
– **ST_GiterateExitPair4**: If the body t can be reduced to a value, then assign a $unit$ to the generator's reference and return a $unit$.

Reduction:

$$\frac{\texttt{t1 / st} \longrightarrow \texttt{t1' / st'}}{\texttt{giterate t1 t2 / st} \longrightarrow \texttt{giterate t1' t2 / st'}}\ (\text{ST\_Giterate1})$$

$$\frac{\texttt{t2 / st} \longrightarrow \texttt{t2' / st'}}{\texttt{giterate v1 t2 / st} \longrightarrow \texttt{giterate v1 t2' / st'}}\ (\text{ST\_Giterate2})$$

```
------------------------------------ (ST_GiterateExitPair1)
giterate (loc l) (abs "_" Unit t) / st
    --> giterate (loc l) t / st


------------------------------------ (ST_GiterateExitPair2)
giterate (loc l) (pair (yield v1) v2) / st
    --> [_:=assign (loc l) (ref v2)]v1 / st


------------------------------------ (ST_GiterateExitPair3)
giterate (loc l) (yield v1) / st
    --> [_:=assign (loc l) unit]v1 / st

------------------------------------ (ST_GiterateExitPair4)
giterate (loc l) v1 / st
    --> [_:=assign (loc l) unit]unit / st
```

The *gnext* takes a iterable object's reference and returns the next item of the object.

– **ST_Gnext2**: If the t is a reference term, then reduce to a *giterate* term.

```
Reduction:

        t / st --> t' / st'
------------------------------------ (ST_Gnext1)
gnext t / st --> gnext t' / st'


------------------------------------ (ST_Gnext2)
gnext (loc L) / st
--> giterate (loc L) !(loc L) / st'
```

5 Subtyping

Subtyping is a form of type polymorphism that plays a fundamental role in object-oriented languages. In this section, I will introduce the subtyping systems for my language.

5.1 Structural Rules

– **T_Trans**: S is a subtype of T if only if S is a subtype of U and U is a subtype of T.
– **T_Refl**: A type T is always a subtype of itself.

Typing:

$$\frac{\texttt{S <: U} \qquad \texttt{U <: T}}{\texttt{S <: T}} \text{ (T\_Trans)}$$

$$\frac{}{\texttt{T <: T}} \text{ (T\_Refl)}$$

5.2 Top

S_Top: Every type is a subtype of Top.

Typing:

$$\frac{}{\texttt{S <: Top}} \text{ (S\_Top)}$$

5.3 STLC

S_Arrow: $S_1 \to S_2$ is a subtype of $T_1 \to T_2$ if only if T_1 is a subtype of S_1 and S_2 is a subtype of T_2.

Typing:

$$\frac{\texttt{T1 <: S1} \qquad \texttt{S2 <: T2}}{\texttt{S1 -> S2 <: T1 -> T2}} \text{ (S\_Arrow)}$$

5.4 Pair

S_Prod: $S_1 * S_2$ is a subtype of $T_1 * T_2$ if only if S_1 is a subtype of T_1 and S_2 is a subtype of T_2.

Typing:

$$\frac{\texttt{S1 <: T1} \qquad \texttt{S2 <: T2}}{\texttt{S1 * S2 <: T1 * T2}} \text{ (S\_Prod)}$$

5.5 Reference

S_Ref: $Ref\ S$ is a subtype of $Ref\ T$ if only if S is a subtype of T.

Typing:

$$\frac{\texttt{S <: T}}{\texttt{Ref S <: Ref T}} \text{ (S\_Ref)}$$

5.6 Generator

S_Itr: $Itr\ S$ is a subtype of $Itr\ T$ if only if S is a subtype of T..

Typing:

$$\frac{\texttt{S <: T}}{\texttt{Itr S <: Itr T}}\ (\texttt{S\_Itr})$$

6 Typing

The type system is used to assign a property called type to various terms in order to remove stuck terms. In my language, the types such as *Arrow*, *Nat*, *Bool*, *Unit*, and *Prod* are similar to the types in chapter *MoreStlc*. I also added a new type *Itr* to identify the iterable term such as *gen*. The term passing to a *next* term must be a iterable term.

6.1 Sequence

T_Seq: If the t_2 has type T_2, then the $seq\ t_1\ t_2$ would have type T_2.

Typing:

$$\frac{\texttt{Gamma; ST |- t2 : T2}}{\texttt{Gamma; ST |- seq t1 t2 : T2}}\ (\texttt{T\_Seq})$$

6.2 Reference

- **T_Loc**: If L is less than the length of type state, then the $loc\ L$ has type $Ref\ T$, where T is the located at the Lth cell of the type state.
- **T_Ref**: If t_1 has type T_1, then $ref\ t_1$ has type $Ref\ T_1$.
- **T_Deref**: If t_1 has type $Ref\ T_1$, then $!t_1$ has type T_1.
- **T_Assign**: If t_1 has type $Ref\ T_1$ and t_2 has type T_1, then t_2 has type $Unit$.

Typing:

$$\frac{\texttt{L < |ST|}}{\texttt{Gamma; ST |- loc L : Ref (lookup L ST)}}\ (\texttt{T\_Loc})$$

$$\frac{\texttt{Gamma; ST |- t1 : T1}}{\texttt{Gamma; ST |- ref t1 : Ref T1}}\ (\texttt{T\_Ref})$$

$$\frac{\texttt{Gamma; ST |- t1 : Ref T1}}{\texttt{Gamma; ST |- !t1 : T1}}\ (\texttt{T\_Deref})$$

$$\frac{\texttt{Gamma; ST |- t1 : Ref T1} \quad \texttt{Gamma; ST |- t2 : T1}}{\texttt{Gamma; ST |- t1 := t2 : Unit}}\ (\texttt{T\_Assign})$$

6.3 Let

T_Let: The *let* $x\ t_1\ t_2$ term has a type T_2 if only if that 1) t_1 has a type T_1 and 2) t_2 has a type T_2 in the $Gamma$ where the x has type T_1 in.

Typing:

$$\frac{\texttt{Gamma; ST |- t1 : T1} \quad \texttt{x |-> T1; Gamma; ST |- t2 : T2}}{\texttt{Gamma; ST |- let x=t1 in t2 : T2}}\ (\texttt{T\_Let})$$

6.4 Fix

T_Fix: The *fix* t_1 term has a type T_1 if only if that t_1 has a type $T_1 \to T_1$.

Typing:

$$\frac{\texttt{Gamma; ST |- t1 : T1 -> T1}}{\texttt{Gamma; ST |- fix t1 : T1}}\ (\texttt{T\_Fix})$$

6.5 While

T_While: The *while* $t_1\ t_2$ term has a type $Unit$ if only if that the predicate function t_1 has a type $T \to Bool$ and the body function t_2 has a type $T \to Unit$.

Typing:

$$\frac{\texttt{Gamma; ST |- t1 : (Ref T) -> Bool} \quad \texttt{Gamma; ST |- t2 : (Ref T) -> Unit}}{\texttt{Gamma; ST |- while t1 t2 : Unit}}\ (\texttt{T\_While})$$

6.6 Generator

- **T_Gen**: The *gen* $t_1\ t_2$ term has a type $Itr\ T_1$ if only if that t_1 is a generator function with type $Arrow\ T_1\ (Ref\ (Arrow\ Unit\ T_2))$ and t_2 is an input with type T_1.
- **T_Gyield**: The *yield* t has a type T if only if that t has type T.
- **T_Giterate**: The *giterate* $t_1\ t_2$ has type T if only if that t_1 is a iterable object's reference with type $Itr\ T$ and t_2 has a iterable object.
- **T_Gnext**: The *next* t has a type T if only if that t is a iterable object's reference with type $Itr\ T$.

Typing:

$$\frac{\texttt{Gamma; ST |- t1 : (Arrow T1 (Ref (Arrow Unit T2)))} \quad \texttt{Gamma; ST |- t2 : T1}}{\texttt{Gamma; ST |- gen t1 t2 : Itr T1}}\ \text{(T\_Gen)}$$

$$\frac{\texttt{Gamma; ST |- t : T}}{\texttt{Gamma; ST |- gyield t : T}}\ \text{(T\_Gyield)}$$

$$\frac{\texttt{Gamma; ST |- t1 : (Itr T)} \quad \texttt{Gamma; ST |- t2 : (Arrow Unit T')}}{\texttt{Gamma; ST |- giterate t1 t2 : T}}\ \text{(T\_Giterate)}$$

$$\frac{\texttt{Gamma; ST |- t : (Itr T)}}{\texttt{Gamma; ST |- next t : T}}\ \text{(T\_Next)}$$

6.7 Subsumption

T_Sub: The term t has type T if only if t has type S and S is subtype of T.

Typing:

$$\frac{\texttt{Gamma; ST |- t : S} \quad \texttt{S <: T}}{\texttt{Gamma; ST |- t : T}}\ \text{(T\_Sub)}$$

7 Conclusion

In this project, I extended lambda calculus with the generator feature to improve the memory efficiency of the iteration operation in Coq. The key idea to implement this feature is that, I defined a *yield* term to trigger the *exit* point and re-*entry* point in the reduction of sequences in a function program. In this way, we can pause and resume a function program in order to iterate each generated item with constant space instead of storing all of them into a list. A generator feature demo is provided in both the Coq code file and below by comparing the ordinary function *firstn* and the generator functionn *gfirstn*.

```
firstn = \n:Nat.
         let num = ref 0 in
         let nums = ref (nil Nat) in
             while P B num;
             !nums

P = \num:(Ref Nat). !num < n
```

```
B = \num:(Ref Nat).
    nums := !nums.append(!num);
    num := succ(!num)
```

With the generator feature, we can not only utilize the iterative functionality in less code, but also access it through efficient memory management.

```
gfirstn = \n:Nat.
          let num = ref 0 in
              ref (\_:Unit (while P B num)

P = \num:(Ref Nat). !num < n
B = \num:(Ref Nat).
    yield !num;
    num := succ(!num)
```

Use *next* function to iterate each item of the generator.

```
let g = gen gfirstn 10 in
let x = next(g) in
let x = next(g) in
    (x, y);
```

However, the generator cannot replace the list-based iteration due to the high cost in generating items. If we need to reuse the generated values multiple times, then the usage of a list is reasonable.

References

1. AshishSingh: Generators. https://wiki.python.org/moin/Generators Accessed: 2018-03-08.
2. Abbas, S.K.: The magic behind python generator functions. https://hackernoon.com/the-magic-behind-python-generator-functions-bc8eeea54220 Accessed: Aug 1, 2017.