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].

```python
# 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.
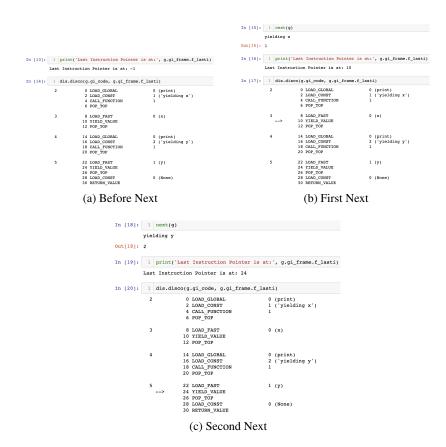
```
In [15]:   1  next(g)

           yielding x
Out[15]: 1
```

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

           Last Instruction Pointer is at: -1
```

```
In [16]:   1  print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)

           Last Instruction Pointer is at: 10
```

```
In [14]:   1  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
```

```
In [17]:   1  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

(b) First Next

```
In [18]:   1  next(g)

           yielding y
Out[18]: 2
```

```
In [19]:   1  print('Last Instruction Pointer is at:', g.gi_frame.f_lasti)

           Last Instruction Pointer is at: 24
```

```
In [20]:   1  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. When the program flow encounters a *yield* term, the term is 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 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
| gyield t1' => pair t1' (abs "_" Unit t2)
| _ => app (abs "_" Unit t2) t1
end.
```

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

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

However, I implemented a sequence term *seq* in my project instead of the *tseq* function for easing sequence concatenation.

2.2   Yield in Iteration

To make sure the *tseq* function would work in a iteration, 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 this way, there is an *exit* pair returned in each *while* iteration. The corresponding proof is provided for the *while* term in the operation semantic section.

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 will be turned to 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 want 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:

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

I defined a *gen* term to construct a iterable generator object in type (Itr T) by taken a generator function and a corresponding input. To iterate each item returned at the *yield* exit point of a *gen* term, I implemented a *next* function shown as below. Here, the x, y are the first two items returned from the generator g.

```
1 let g = gen G x in
2 let x = next(g) in
3 let y = next(g) in
4 (x, y)
```

In the following sections, I provide the approaches including syntax, operation seman-
tics, 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.

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
   |  next  t                   next  function

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

V  ::=                          Values
   |  ...
   |  loc  l                    reference  cell  location
   |  yield  t                  yeild  term
```

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 *next* takes a iterable term.

For the other required terms, I am not going to show more details here since they are
similar to those terms in chapter *MoreStlc*. The *number* is required to count down the
rest iterations. The *boolean* is required for conditional statement. The *pair* is required

for return pair values when the execution encounters a exit point in a generator function. The *fix* is used for build *while* term.

Types: The $next$ function should only take iterable terms. So, I added a iterator type to identify the iterable generator. And other types like Arrow, Nat, Bool, List, Unit, and Prod are similar to the terms in chapter *MoreStlc*.

Values: Both *loc l* and *yield t* are considered as value terms without any further reduction.

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 can be constructed by taking two terms, where both could be *seq* terms. I also defined three help functions: *yield_tm* for checking the *yield* term, *seq_tm* for check the *seq* term, and *seqCat* for the concatenation of two terms. For the reduction, **ST_-Seq1**: If t1 is a *seq* term, then concatenate t1 with t2. **ST_Seq2**: If t1 is not a *seq* term, then reduce t1 to t1'. **ST_Seq3**: If v1 is a *yield* term, then return a *exit* pair. **ST_Seq4**: If v1 is not a *yield* term, then return a t2.

```
Inductive yield_tm : tm -> Prop :=
| Tgyield : forall t, yield_tm (gyeild t).

Inductive seq_tm : tm -> Prop :=
| Tseq : forall t1 t2, seq_tm (seq t1 t2).

Fixpoint seqCat (s : tm) (t : tm) : tm :=
match s with
| seq t1 t2 => seq t1 (seqCat t2 t)
| t' => seq t' t
end.
```

Reduction:

$$\frac{seq\_tm\ t1}{seq\ t1\ t2\ /\ st\ \longrightarrow\ seqCat\ t1\ t2\ /\ st}\ (ST\_Seq1)$$

$$\frac{not\ (seq\_tm\ t1)\quad t1\ /\ st\ \longrightarrow\ t1'\ /\ st'}{seq\ t1\ t2\ /\ st\ \longrightarrow\ seq\ t1'\ t2\ /\ st'}\ (ST\_Seq2)$$

$$yield\_tm\ v1$$

$$\frac{}{\begin{array}{l} \text{seq v1 t2 / st} \\ \quad \text{--> pair v1 (\\_:Unit t2) / st} \end{array}} \quad (\text{ST\_Seq3})$$

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

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 v2.

```
Definition store := list tm.
Definition store_lookup (n:nat) (st:store) := nth n st unit.
```

Reduction:

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

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

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

$$\frac{\text{l < |st|}}{\text{!(loc l) / st --> lookup l st / st}} \quad (\text{ST\_DerefLoc})$$

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

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

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

4.3 Let

ST_LetValue: If v1 is a value, then substitute x with v1 in t2.

Reduction :

$$\frac{\texttt{t1 / st --> t1' / st'}}{\begin{array}{c}\texttt{let x=t1 in t2 / st}\\\texttt{--> let x=t1' in t2 / st'}\end{array}} \quad (\,\mathrm{ST\_Let1}\,)$$

$$\frac{}{\begin{array}{c}\texttt{let x=v1 in t2 / st}\\\texttt{--> [x:=v1]t2 / st'}\end{array}} \quad (\,\mathrm{ST\_LetValue}\,)$$

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 follow:

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

Reduction :

$$\frac{\texttt{t1 / st --> t1' / st'}}{\texttt{fix t1 / st --> fix t1' / st'}} \quad (\,\mathrm{ST\_Fix1}\,)$$

$$\frac{}{\begin{array}{c}\texttt{fix (\textbackslash xf:T1.t2) / st -->}\\\texttt{[xf:=fix (\textbackslash xf:T1.t2)] t2 / st}\end{array}} \quad (\,\mathrm{ST\_FixAbs}\,)$$

4.5 While

Since *while* loop can be implemented in recursion as shown below, assume x has a reference type, 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.

```
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 (\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, the *exit* pair would be triggered by the reduction of *seq* even a *yield* term is nested in a *while* term that itself is also a term in a sequence.

Reduction:

$$\frac{t1 \; / \; st \; --> \; t1' \; / \; st'}{while \; t1 \; t2 \; / \; st \; --> \; while \; t1' \; t2 \; / \; st'} \; (ST\_While1)$$

$$\frac{t2 \; / \; st \; --> \; t2' \; / \; st'}{while \; v1 \; t2 \; / \; st \; --> \; while \; v1 \; t2' \; / \; st'} \; (ST\_While2)$$

$$\frac{}{\begin{array}{l} while \; (\backslash x1:(Ref \; T). \; p) \; (\backslash x2:(Ref \; T). \; b) \; / \; st \\ --> \; tfix \; (\backslash f:(Ref \; T)->Unit. \\ \qquad (\backslash x:(Ref \; T). \; (if \; \backslash x1:(Ref \; T). \; p \; x \\ \qquad\qquad\qquad then \; seq \; (\backslash x2:T. \; b \; x) \; (f \; x) \\ \qquad\qquad\qquad else \; unit))) \; / \; st' \end{array}} \; (ST\_WhileFix)$$

4.6 Generator

ST_Gen3: The *gen* term takes a generator function and a corresponding input and returns a reference of a iterable object. **ST_Gnext2**: The *next* function takes a *gen* term and reduces it into a pair, where the first term the generator's reference and the second is the value under it. **ST_Gnext3**: If the second term of pair is a unit function, apply and reduce the body term t with **ST_Gnext1**. **ST_Gnext4**: If the second term is a *exit* pair, update the generator's reference with the new generator v2 and return v1. **ST_Gnext5**: If the second term is a *yield* term, update the generator's reference with unit and return v.

Reduction:

$$\frac{t1 \; / \; st \; --> \; t1' \; / \; st'}{gen \; t1 \; t2 \; / \; st \; --> \; gen \; t1' \; t2 \; / \; st'} \; (ST\_Gen1)$$

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

$$\frac{}{gen \ (\backslash x:T. \ g) \ v \ / \ st \ \longrightarrow \ [x:=v]g \ / \ st} \ (ST\_Gen3)$$

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

$$\frac{t \ / \ st \ \longrightarrow \ t' \ / \ st'}{gnext \ t \ / \ st \ \longrightarrow \ gnext \ t' \ / \ st'} \ (ST\_Gnext1)$$

$$\frac{}{\begin{array}{l} gnext \ (loc \ l) \ / \ st \\ \quad \longrightarrow \ gnext \ (pair \ (loc \ l) \ (deref \ (loc \ l))) \ / \ st \end{array}} \ (ST\_Gnext2)$$

$$\frac{}{\begin{array}{l} gnext \ (pair \ (loc \ l) \ (abs \ "\_" \ Unit \ t)) \ / \ st \\ \quad \longrightarrow \ gnext \ (pair \ (loc \ l) \ t) \ / \ st \end{array}} \ (ST\_Gnext3)$$

$$\frac{}{\begin{array}{l} gnext \ (pair \ (loc \ l) \ (pair \ (yield \ v1) \ v2)) \ / \ st \\ \quad \longrightarrow \ [\_:=assign \ (loc \ l) \ (ref \ v2)]v1 \ / \ st \end{array}} \ (ST\_Gnext4)$$

$$\frac{}{\begin{array}{l} gnext \ (pair \ (loc \ l) \ (yield \ v)) \ / \ st \\ \quad \longrightarrow \ [\_:=(assign \ (loc \ l) \ unit)]v \ / \ st \end{array}} \ (ST\_Gnext5)$$

5 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*, *List*, *Unit*, and *Prod* are similar to the types in chapter *MoreStlc*. They are used to specify the types for lambda calculus, natural number, boolean, list, unit, and pair. I also added a new type *Itr T* to identify the iterable term such as *gen*. The first term of the *next* function must be a iterable term.

5.1 Let

The *let* term has a type $T2$ if only if that 1) $t1$ has a type $T1$ and 2) $t2$ has a type $T2$ in the $Gamma$ where the x has type $T1$ in.

Typing:

$$
\frac{\begin{array}{c} \text{Gamma } |- \text{ t1 } \backslash \text{in T1} \\ \text{x } |->\text{T1; Gamma } |- \text{ t2 } \backslash \text{in T2} \end{array}}{\text{Gamma } |- \text{ let x=t1 in t2 } \backslash \text{in T2}} \quad (\text{T\_Let})
$$

5.2 Fix

The *fix* term has a type $T1$ if only if that $t1$ has a type $T1 \to T1$.

Typing:

$$
\frac{\text{Gamma } |- \text{ t1 } \backslash \text{in T1 } -> \text{ T1}}{\text{Gamma } |- \text{ fix t1 } \backslash \text{in T1}} \quad (\text{T\_Fix})
$$

5.3 While

The *while* term has a type T if only if that the predicate function $t1$ has a type $T \to Bool$ and the body function $t2$ has a type $T \to T$.

Typing:

$$
\frac{\begin{array}{c} \text{Gamma } |- \text{ t1 } \backslash \text{in T } -> \text{ Bool} \\ \text{Gamma } |- \text{ t2 } \backslash \text{in T } -> \text{ T} \end{array}}{\text{Gamma } |- \text{ while t1 t2 } \backslash \text{in T}} \quad (\text{T\_While})
$$

5.4 Generator

The *gen* term has a type $(Itr\ T)$ if only if that the predicate function $t1$ has a type $T \to Bool$ and the body function $t2$ has a type $T \to T$. And the *next* term has a type $(T * T)$ if only if the $t1$ iterable with a type $(Itr\ T)$ and the body function $t2$ has a type T.

Typing:

$$
\frac{\begin{array}{c} \text{Gamma } |- \text{ t1 } \backslash \text{in T } -> \text{ Bool} \\ \text{Gamma } |- \text{ t2 } \backslash \text{in T } -> \text{ T} \end{array}}{\text{Gamma } |- \text{ gen t1 t2 } \backslash \text{in (Itr T)}} \quad (\text{T\_Gen})
$$

$$\frac{\begin{array}{c} \text{Gamma} \vdash t1 \in (Itr\ T) \\ \text{Gamma} \vdash t2 \in T \end{array}}{\text{Gamma} \vdash next\ t1\ t2 \in (T * T)} \quad (T\_Next)$$

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.