

Continuations and Stackless Python

Christian Tismer

Presented by
Prashanth Pappu

Sneak Peek

- Paper presents ideas to
 - Implement co-routines and generators using the concept of “continuations” in Python.
 - Somewhere down the line, author realizes that machine-independent implementation of “continuations” requires a “stackless python”
 - So, the author presents ideas to make python “stackless”

A better Sales Pitch

- Stackless Python requires smaller ‘memory footprints’ and can enable use of Python on devices like handhelds.
- Also, a stackless python enables efficient implementation of additional constructs like co-routines and generators.

Continuations

- Every single line of a code has a continuation that represents the entire future of execution of the program.
- `x = 2; y = x + 1; z = x * 2;`
- Continuation of `x = 2` is `y = x + 1; z = x * 2;`
- Example from the paper that I don't really understand.

```
Def looptest(n):  
    this = continuation.current();  
    k = this.update(n)  
    if k:  
        this(k-1)  
    else:  
        del this.link
```

- Note that the above continuations can accept parameters and return values.

Continuations

- A continuation at any point in the execution of the program is an abstraction of the *rest of the program*.

```
Main()
{
    p('a');
    p('b');
}

Void p(char x)
{
    putchar(x);
}
```

Sample Program

```
Main()
{
    p('a', &&L1);
    L1: p('b', &&L2);
    L2: exit;
}

Void p(char x, void *k)
{
    putchar(x);
    goto *k;
}
```

Using Goto

```
Main()
{
    p('a', &L1);
}

Void p(char x, void (*k) ())
{
    putchar(x);
    (*k)();
}

Void L1(){p(b,&exit);}
```

Non-returning functions

What's the idea?

- Continuations are a primitive form of capturing and manipulating *the control flow of a program*.
- It can essentially be used to synthesize any form of control flow.
- More relevantly, shows that call-return primitives with the use of stack are not necessary for control flow.
- Also more concretely, a continuation at any point in a code represents the 'state' of the execution (locals, globals, program and stack counter) -- a Python frame (actually, more than just a frame).

Generators

- Generator

```
Node Advance()  
{  
    Node n = list->first();  
    do{  
        yield n;  
    }while((n=n->next) != NULL);  
    yield NULL;  
}
```

- Generators resume exactly at the point where it last left off.
- Generators can also be seen as iterators.

Coroutines

- Consider two pieces of code,

```
While(1){  
    len = getint();  
    char = getchar();  
    for(i=0;i<len;i++){  
        emit(char);  
    }  
}
```

Decompression code

```
While(1){  
    c = getchar();  
    /* parse etc */  
}
```

Parser code

- How do we get the two pieces of code to communicate.
- One way would be to turn them into functions.
- Rewritten pieces of code can often be much harder to read and 'ugly'
- Knuth's solution, to not regard one of them as caller and callee but treat them as equals (coroutines)

Generators

- Implementing generators and coroutines using continuations.

```
Def put(self,value)
{
    self.producer = get_caller()
    self.consumer(value)
}
```

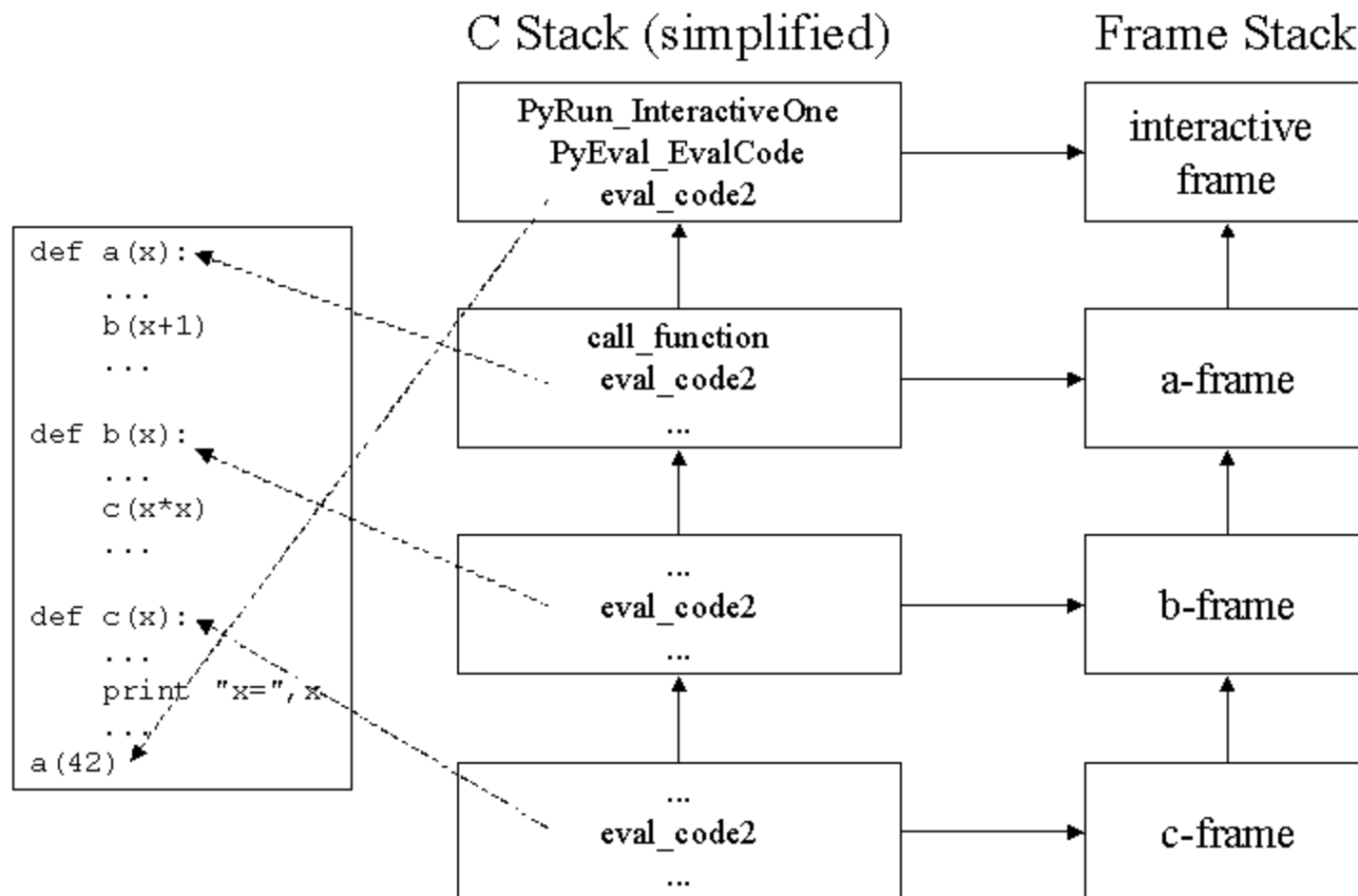
```
Def get(self)
{
    self.consumer = get_caller()
    self.producer()
}
```

- `get()` and `put()` jump directly into the 'context' of the consumer/producer function.
- Jumps are expressed as calls of continuations, which leave current context and jump over to the context that is stored in the continuation object.
- Important : Note how continuations act as building blocks for new control structures.

Stackless Python

- Totally orthogonal topic which concentrates on implementing continuations in Python.
- Current stackfull Python doesn't enable 'jumping to any context' or simple implementation of continuations.
- Current implementations of generators and coroutines are of low performance – either use threads, limited by python's recursive interpreter layout which prevents liberal switching between frames.
- Continuations cannot be implemented without stackless python in a machine-independent manner.
- Some notes on Python code interpreter written in 'C'.
 - Every python action is somehow performed by 'C code'.
 - Every piece of Python code invokes a new interpreter loop.

Standard „stackfull“ Python



Why C stack should vanish

- C stack limits recursion depth.
 - Fixed size?
- C stack holds references to objects and execution state.
 - Limits possibility of out of order execution.
- Removing C stack is cheap.
 - Paper shows that minimal changes were made.
 - Wouldn't go into these details (uninteresting)
- Coroutines can be 'incredibly' fast.
 - Built-in function call cheaper than Python function.
 - Might enable their use in many problems.
- PERL no longer has a C stack!
 - PERL folk are apparently smarter and if they have got rid of their stack author sees no reason why python buffs can't follow suit.

How?

- Time of Frame Execution
 - Old paradigm – build a frame, put params in place, run interpreter, wait for return.
 - Transformation : Running Frames in correct order does not mean we need to call interpreter from current nesting level.
 - Avoid any C-stack related post-processing.
- Lifetime of Parameters
 - A reference to params is put in frame and removed when the frame is disposed as opposed to 'C interpreter' removing the reference.
- Third system state returned by frames to control unwinding of C stack, apart from the usual `Null` and `PyObject`.

Essential Idea (as I understood it)

- Tail recursion

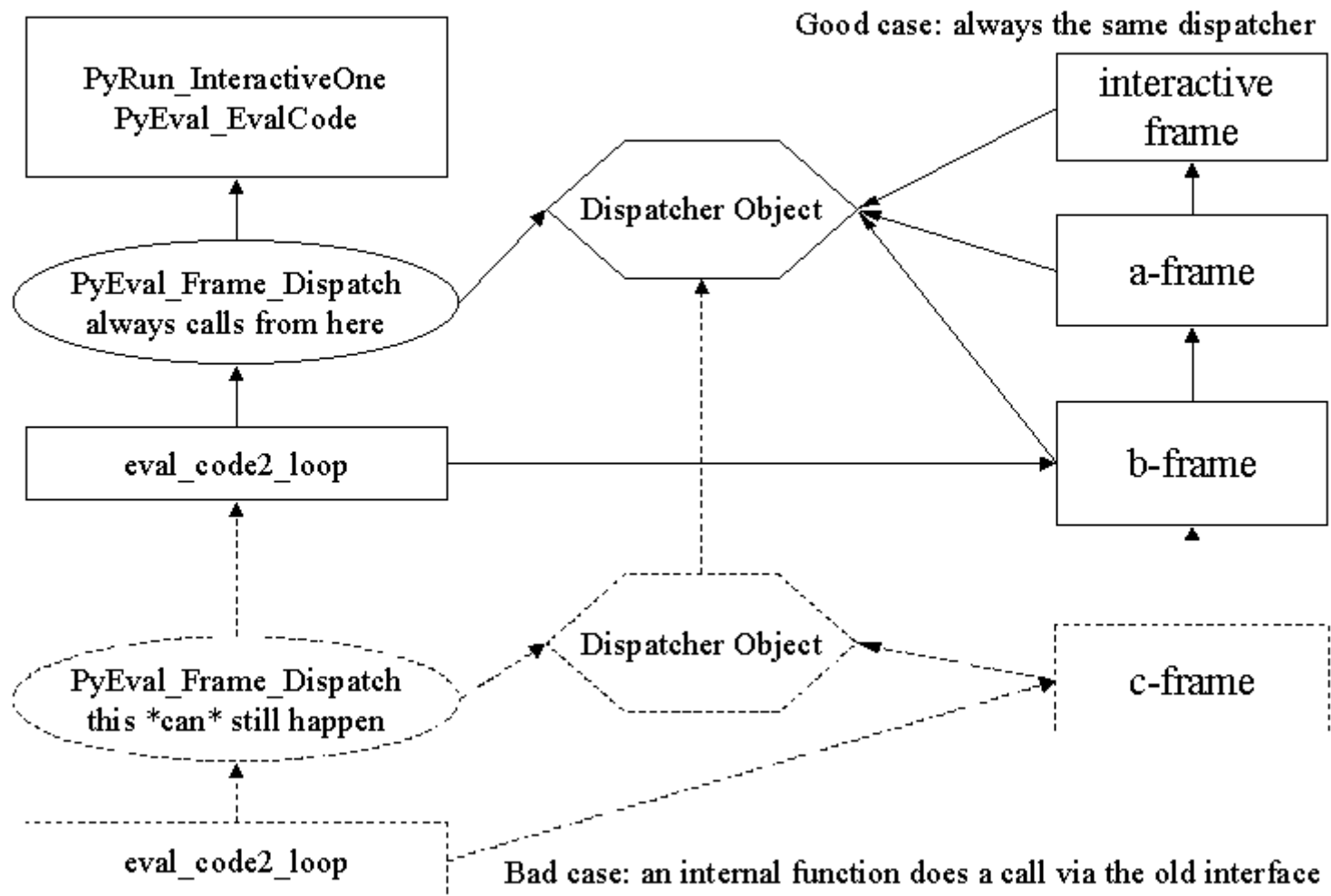
$$f(x+1) = a + f(x);$$

- If function body enables 'tail recursion' no 'state' is necessary for function evaluation.

- $f(n) = a + (a + f(n-2)) = 2a + f(n-2) !$

- $f(x+1) = f(x) + a$ gives the same result but requires state to be stored per function call.

Stackless Python



Last Slide

- “Changing python to become non-recursive was considered a major difficult task. Actually, it was true and I had to change truth before I could continue”.
- Paper helps appreciate what continuations are etc, but also has a good dose of Python interpreter code.
- There are a lot of C API extension, compatibility issues and related code examples which I have conveniently ignored.